UNICOS® System Calls Reference Manual

SR–2012 10.0

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

New Features

UNICOS® System Calls Reference Manual SR–2012 10.0

The *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012, incorporates the following technical changes for the UNICOS 10.0 release. In addition, miscellaneous technical, editorial, and formatting changes have been made throughout the manual.

The following system calls are new:

`getash`(2)	Gets an array session handle
`newarraysess`(2)	Starts a new array session
`setash`(2)	Sets an array session handle
`statvfs`(2)	Gets file system information
`syssgi`(2)	Provides a system interface to Silicon Graphics workstations

The following system calls have been updated:

`acctctl`(2)	Supports the socket accounting feature, which tracks network usage from the perspective of sockets
`limit`(2)	Contains a new limit value of `NO_CORE_FILES` for the `L_CORE` resource to disable core file creation
`quotactl`(2)	Supports the optional aggregate quota feature.

Record of Revision

Version	*Description*
1.0	March 1996 Original Printing. Documentation to support the UNICOS release 1.0 running on Cray computer systems.
1.1	June 1986 Online documentation only to support the UNICOS release 1.1 running on Cray computer systems.
2.0	September 1986 Documentation to support the UNICOS release 2.0 running on Cray computer systems.
3.0	July 1987 Documentation to support the UNICOS release 3.0 running on Cray computer systems.
4.0	July 1988 Documentation to support the UNICOS release 4.0 running on Cray computer systems.
5.0	January 1989 Documentation to support the UNICOS release 5.0 running on Cray computer systems.
6.0	February 1991 Documentation to support the UNICOS 6.0 release running on all Cray Research systems.
7.0	September 1992 Documentation to support the UNICOS 7.0 release running on all Cray Research systems.
8.0	January 1994 Documentation to support the UNICOS 8.0 release running on all Cray Research systems.
9.0	August 1995 Documentation to support the UNICOS 9.0 release running on all Cray Research systems.

9.3	August 1997 Documentation to support the UNICOS 9.3 release running on all Cray Research systems.
10.0	November 1997 Documentation to support the UNICOS 10.0 release running on Cray Research systems. The New Features page provides detailed information about the system call changes documented in this manual.

Preface

This publication documents the UNICOS 10.0 release running on Cray Research systems and supplements the information contained in the other manuals in the UNICOS documentation set. It describes the UNICOS system calls, which access the services provided by the system kernel.

This is a reference manual for UNICOS programmers. Readers should have a working knowledge of either the UNICOS or the UNIX operating system.

Related publications

The following man page manuals contain additional information that may be helpful.

> **Note:** For the UNICOS 10.0 release, man page reference manuals are not orderable in printed book form. Instead, they are available as printable PostScript files provided on the same DynaWeb CD as the rest of the supporting documents for this release. Individual man pages are still available online and can be accessed by using the `man`(1) command.

- *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011
- *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014
- *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022
- *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

The following ready references are available in printed form from the Distribution Center:

- *UNICOS User Commands Ready Reference*, Cray Research publication SQ–2056
- *UNICOS System Libraries Ready Reference*, Cray Research publication SQ–2147
- *UNICOS System Calls Ready Reference*, Cray Research publication SQ–2215
- *UNICOS Administrator Commands Ready Reference*, Cray Research publication SQ–2413

The following manuals are also referenced on man pages in this document:

- *Tape Subsystem User's Guide*, Cray Research publication SG–2051
- *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165
- *General UNICOS System Administration*, Cray Research publication SG–2301
- *UNICOS Networking Facilities Administrator's Guide*, Cray Research publication SG–2304

Ordering Cray Research publications

The *User Publications Catalog*, Cray Research publication CP–0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907, or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:

1	User commands
1B	User commands ported from BSD
2	System calls
3	Library routines, macros, and opdefs
4	Devices (special files)
4P	Protocols
5	File formats
7	Miscellaneous topics
7D	DWB-related information
8	Administrator commands

Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.

variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

The following machine naming conventions may be used throughout this document:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems.
Cray MPP systems	All configurations of the CRAY T3D series. The UNICOS operating system is not supported on CRAY T3E systems. CRAY T3E systems run the UNICOS/mk operating system.

All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release.

The default shell in the UNICOS and UNICOS/mk operating systems, referred to in Cray Research documentation as the *standard shell*, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS and UNICOS/mk operating systems also support the optional use of the C shell.

Cray UNICOS Version 10.0 is an X/Open Base 95 branded product.

Man page sections

The entries in this document are based on a common format. The following list shows the order of sections in an entry and describes each section. Most entries contain only a subset of these sections.

Section heading	Description
NAME	Specifies the name of the entry and briefly states its function.
SYNOPSIS	Presents the syntax of the entry.
IMPLEMENTATION	Identifies the Cray Research systems to which the entry applies.
STANDARDS	Provides information about the portability of a utility or routine.
DESCRIPTION	Discusses the entry in detail.
NOTES	Presents items of particular importance.
CAUTIONS	Describes actions that can destroy data or produce undesired results.
WARNINGS	Describes actions that can harm people, equipment, or system software.
ENVIRONMENT VARIABLES	Describes predefined shell variables that determine some characteristics of the shell or that

	affect the behavior of some programs, commands, or utilities.
RETURN VALUES	Describes possible return values that indicate a library or system call executed successfully, or identifies the error condition under which it failed.
EXIT STATUS	Describes possible exit status values that indicate whether the command or utility executed successfully.
MESSAGES	Describes informational, diagnostic, and error messages that may appear. Self-explanatory messages are not listed.
ERRORS	Documents error codes. Applies only to system calls.
FORTRAN EXTENSIONS	Describes how to call a system call from Fortran. Applies only to system calls.
BUGS	Indicates known bugs and deficiencies.
EXAMPLES	Shows examples of usage.
FILES	Lists files that are either part of the entry or are related to it.
SEE ALSO	Lists entries and publications that contain related information.

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  `publications@cray.com`

- Contact your customer service representative and ask that an SPR or PV be filed. If filing an SPR, use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

1–800–950–2729 (toll free from the United States and Canada)

+1–612–683–5600

- Send a facsimile of your comments to the attention of “Software Publications Group” in Eagan, Minnesota, at fax number +1–612–683–5599.

We value your comments and will respond to them promptly.

CONTENTS

NAME

`intro` – Introduces system calls and error numbers

SYNOPSIS

```
#include <errno.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This manual describes all UNICOS system calls. `intro`(2) contains sections on the following:

- Security privilege information
- Socket system calls
- Errors, including a listing of all error numbers

Security Privilege Information

If your UNICOS system is using the default privilege assignment lists (PALs), many of the UNICOS system calls expect that the calling process has certain privileges effective in order for the system call to execute correctly.

The man page for each affected UNICOS system call lists the privileges associated with the call. Also included is a description of what tasks or functions the associated privileges allow the system call to perform. For a list of the privileges and a general description of each privilege used on a UNICOS system, see the Security section in *General UNICOS System Administration*, Cray Research publication SG–2301.

Socket System Calls

The transport-level protocols, `tcp`(4P) and `udp`(4P), along with the network level protocol `ip`(4P), are described in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.

A socket is a general network interface that provides programs with a uniform view of network protocol suites. The program relates to the sockets rather than the protocol; therefore, the program can use any network protocol suite (such as TCP).

Certain semantics of the basic socket abstractions are protocol-specific. Each protocol is expected to support the basic model for its particular socket type, but may, in addition, provide nonstandard facilities or extensions to a mechanism. For example, a protocol supporting the `SOCK_STREAM` abstraction may allow more than 1 byte of out-of-band data to be transmitted per out-of-band message.

Sockets using the TCP protocol are either *active* or *passive*. Active sockets initiate connections to passive sockets. By default, TCP sockets are created active; to create a passive socket, the `listen`(2) system call must be used after binding the socket with the `bind`(2) system call. Only passive sockets may use the `accept`(2) call to accept incoming connections. Only active sockets may use the `connect`(2) call to initiate connections.

The Internet family, `inet`(4P), provides protocol support for the `SOCK_STREAM`, `SOCK_DGRAM`, and `SOCK_RAW` socket types. Transmission Control Protocol (TCP) supports the `SOCK_STREAM` abstraction, while the User Datagram Protocol (UDP) supports the `SOCK_DGRAM` abstraction. A super user may achieve a raw interface to the Internet Protocol (IP) and Internet Control Message Protocol (ICMP) by creating an Internet socket of type `SOCK_RAW`.

A passive socket may underspecify its location to match incoming connection requests from networks. This technique, termed *wildcard addressing*, allows one server to provide service to clients on multiple networks. To create a socket that listens on all networks, the Internet address `INADDR_ANY` must be bound. The TCP port may still be specified at this time; if the port is not specified, the system assigns one. When a connection has been established, the socket's address is fixed by the peer entity's location. The address assigned to the socket is the address associated with the network interface through which packets are being transmitted and received. Usually, this address corresponds to the peer entity's network. UDP supports the `SOCK_DGRAM` abstraction for the Internet protocol family. UDP sockets are connectionless, and they are normally used with the `sendto` (see `send`(2)) and `recvfrom` (see `recv`(2)) calls, although the `connect`(2) call may also be used to fix the destination for future packets (in which case, `recv`(2) and `send`(2) or the `read`(2) or `write`(2) system calls can be used).

A write to a raw socket must not include the IP header at the beginning of the data unless the `IP_HDRINCL` socket option has been set (see `setsockopt`(2)). A read from a raw socket always returns the IP header.

Internet Control Message Protocol (ICMP) sockets are also available through raw sockets. Refer to `icmp`(4P). The files are:

- TCP/IP library routines are in `/lib/libc.a`.
- TCP/IP include files are in the directory `/usr/include`.
- Symbolic names for errors are in `/usr/include/errno.h`.

Errors

Most system calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value, which is almost always −1; the individual descriptions specify the details. An error number is also made available in the `errno` external variable, which is not cleared on successful calls; therefore, it should be tested only after an error has been indicated. Each system call description tries to list all possible error numbers.

The following is a list of the error numbers and their names as defined in the `errno.h` header file. Tape users may receive error codes that are not documented on the system call man pages. For the descriptions of the tape daemon return values, see the *Tape Subsystem User's Guide*, Cray Research publication SG–2051.

`1 EPERM Not owner`
: Typically, this error indicates that you attempted to modify a file in a way not allowed except to its owner or a super user. It is also returned when other users try to perform actions allowed only to super users.

`2 ENOENT No such file or directory`
: Either the specified file does not exist or one of the directories in a path name does not exist.

`3 ESRCH No such process`
: No process can be found corresponding to the process that you specified.

`4 EINTR Interrupted system call`
: An asynchronous signal (such as `interrupt` or `quit`), which you have elected to catch, occurred during a system call. If execution is resumed after processing the signal, it appears that the interrupted system call returned this error condition.

`5 EIO I/O error`
: A physical I/O error has occurred. In some cases, this error may occur on a call following the one to which it actually applies.

`6 ENXIO No such device or address`
: During a read or write on a special file, a subdevice that does not exist or is beyond the limits of the device was referenced.

`7 E2BIG Arg list too long`
: Your argument list to a member of the `exec`(2) family is longer than `NCARGS` bytes.

`8 ENOEXEC Exec format error`
: A request was made to execute a file that, although it has the appropriate permissions, does not start with a valid magic number (see `a.out`(5)).

`9 EBADF Bad file number`
: Either a file descriptor refers to no open file, or a read or write request is made to a file that is open only for writing or reading, respectively.

`10 ECHILD No child processes`
: A `wait`(2) system call was executed by a process that had no existing or unwaited-for child processes.

`11 EAGAIN Resource temporarily unavailable`
The resource is unavailable now; later calls to the same routine may complete normally.

`12 ENOMEM Not enough space`
During an `exec`(2) or `sbreak`(2) system call, a program requested more space than the system could supply. This is not a temporary condition; the maximum space specification is a system parameter.

`13 EACCES Permission denied`
You attempted to access a file in a way not allowed by the protection system.

`14 EFAULT Bad address`
The system encountered a hardware fault in attempting to use an argument of a system call.

`15 ENOTBLK Block device required`
A call specifies something other than a block device where a block device is required (for example, in `mount`(2)).

`16 EBUSY Device busy`
You attempted to mount a device that was already mounted or to dismount a device on which there is an active file (open file, current directory, mounted-on file, or active text segment). The device or resource is currently unavailable.

This error also occurs if you try to enable accounting when it is already enabled or if you issue a `restart`(2) attempt when another job or process in the system is using the *jid* or any *pid* associated with the job (or process) to be restarted.

`17 EEXIST File exists`
A call specifies an existing file in an inappropriate context (for example, `link`).

`18 EXDEV Cross-device link`
You attempted a link to a file on another logical device.

`19 ENODEV No such device`
You attempted to apply an inappropriate system call to a device (for example, to read a write-only device).

`20 ENOTDIR Not a directory`
A call specifies a nondirectory where a directory is required (for example, in a path prefix or as an argument to `chdir`(2)).

`21 EISDIR Is a directory`
You attempted to write on a directory.

`22 EINVAL Invalid argument`
The call contains an argument that is not valid such as the dismounting of a nonmounted device, the mention of an undefined signal in `signal`(2) or `kill`(2), or the reading or writing of a file for which `lseek`(2) has generated a negative pointer. This error is also set by the math functions described in the (3) entries.

`23 ENFILE File table overflow`
The system file table is full and temporarily cannot accept more `open`(2) calls.

`24 EMFILE Too many open files`
No process can have more than `NOFILE` file descriptors open at a time.

`25 ENOTTY Not a typewriter`
You attempted to use an `ioctl`(2) request with a file that is not a character special file.

`26 ETXTBSY Text file busy`
Either you attempted to execute a pure-procedure program that is currently open for writing or reading, or you attempted to open for writing a pure-procedure program that is being executed.

`27 EFBIG File too large`
The size of a file exceeds the maximum file size or the process size set by the `ulimit`(2) system call.

`28 ENOSPC No space left on device`
During a `write`(2) to an ordinary file, the free space left on the device was exhausted.

`29 ESPIPE Illegal seek`
You issued an `lseek`(2) to a pipe. This is not allowed.

`30 EROFS Read-only file system`
You attempted to modify a file or directory on a device mounted as read-only.

`31 EMLINK Too many links`
You attempted to make more than `LINK_MAX` links to a file.

`32 EPIPE Broken pipe`
A write was performed on a pipe for which no process exists to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

`33 EDOM Argument out of domain`
The argument of a function in the math package (3) is out of the domain of the function.

`34 ERANGE Result too large`
The value of a function in the math package (3) cannot be represented within machine precision.

`35 ENOMSG No message of desired type`
This message is used by internal tools. If you receive this message, contact your system administrator.

`36 EIDRM Identifier removed`
This message is reserved for future use. If you receive this message, contact your system administrator.

`37 ECHRNG Channel number out of range`
This message is reserved for future use. If you receive this message, contact your system administrator.

`38 EL2NSYNC Level 2 not synchronized`
This message is reserved for future use. If you receive this message, contact your system administrator.

`39 EL3HLT Level 3 halted`
This message is reserved for future use. If you receive this message, contact your system administrator.

`40 EL3RST Level 3 reset`
This message is reserved for future use. If you receive this message, contact your system administrator.

`41 ELNRNG Link number out of range`
This message is reserved for future use. If you receive this message, contact your system administrator.

`42 EUNATCH Protocol driver not attached`
This message is reserved for future use. If you receive this message, contact your system administrator.

`43 ENOCSI No CSI structure available`
This message is reserved for future use. If you receive this message, contact your system administrator.

`44 EL2HLT Level 2 halted`
This message is reserved for future use. If you receive this message, contact your system administrator.

`45 EDEADLK Deadlock situation detected/avoided`
A deadlock situation was detected and avoided. This error pertains to file and record locking.

`46 ENOLCK No record locks available`
The setting or removing of record locks on a file cannot be accomplished, because no more record lock entries are left in the system.

`47 EINVFS Not allowed on this file system`
An operation was performed on a file system type that does not support that operation.

`50 EFILECH File changed`
Either a file referenced by the restart file has been changed since the restart file was created, or a file residing remotely on a network file system (NFS) has changed.

`51 EFILERM File removed`
Either a file needed for the checkpointing or restarting of a job or process has no links to it, or a file residing remotely on a network file system (NFS) has been removed.

`52 ERFLOCK Recovery of file lock would block`
In `restart`(2), record locks owned by the processes to be restarted could not be recovered, because record locks owned by currently existing processes have one or more of the target file regions already locked.

`53 ENOSDS Unable to recover SDS space`
(Cray PVP systems) During an attempt to recover a job by using `restart`(2), the secondary data segment (SDS) requirement of this job exceeded the current availability of SDS.

`54 EFILESH Fair-share scheduler controls file.`
An unlinked regular file needed for the checkpointing of a job or process is in use by one or more processes outside the set of processes to be checkpointed.

`55 EMALFORMED Malformed process collection`
The target set specified by a `chkpnt`(2) request does not represent a completely contained set. For example, if a process is using a file that is currently opened by more than one process, and if all of the processes that have the file opened are not within the specified process set, this error occurs.

`56 EFOREIGNFS Foreign file system`
An operation that is supported only on local file systems was attempted on a nonlocal (foreign) file system.

`60 EQUSR User file/inode quota limit reached`
A program writing a file under your current user ID has reached a file or inode quota limit. For a temporary solution, remove some of your files so that additional space is available or move the files to a file system that has sufficient quota authorization. For a more permanent solution, contact your system administrator and request additional space.

`61 EQGRP Group file/inode quota limit reached`
A program writing a file under your current group ID has reached a file or inode quota limit. For a temporary solution, remove some of your files so that additional space is available or move the files to a file system that has sufficient quota authorization. For a more permanent solution, contact your system administrator and request additional space.

`62 EQACT Account file/inode quota limit reached`
A program writing a file under your current account ID has reached a file or inode quota limit. For a temporary solution, remove some of your files so that additional space is available or move the files to a file system that has sufficient quota authorization. For a more permanent solution, contact your system administrator and request additional space.

`66 EREMOTE Object is remote`
No explanation is available for this message.

`74 EMULTIHOP Multihop attempted`
No explanation is available for this message.

`75 ESHMA Process has shared memory segment attached`
Process with attached shared memory segments (CRAY T90 series systems only) cannot be checkpointed.

`90 EPROCLIM Process limit exceeded`
This message is returned when a user exceeds the fair-share scheduler process limit.

`91 EMEMLIM Memory limit exceeded`
This message is returned when a user exceeds the fair-share scheduler memory limit.

`92 EDISKLIM Disk limit exceeded`
Your job-based or process disk limit has been exceeded.

`93 ETOOMANYU Too many users`
You exceeded the system compile-time definition of `NUSERS`, which defaults to 200.

`94 ENAMETOOLONG Filename too long`
Either the size of a path name string exceeds the maximum allowed, or a path name component exceeds the maximum and automatic component truncation is not supported for the file system of the file that the component names.

`95 ENOSYS Function not implemented`
An attempt was made to use a function that is not available in this implementation.

`96 ENOTEMPTY Directory not empty`
This error code is defined for compatibility with the POSIX 1003.1 standard, but it is not used.

`97 ERENAMESELF Attempt to rename a link to itself`
This error code is used internally in the `rename`(2) system call. It is never returned to a user program.

`98 ELOOP Too many symbolic links`
Too many symbolic links were encountered when a path name was translated.

UNICOS issues the following TCP/IP network, socket interface error codes:

`128 EWOULDBLOCK Operation would block`
An operation that would cause a process to block was attempted on an object in nonblocking mode (see `ioctl`(2)).

`129 EINPROGRESS Operation now in progress`
An operation that takes a long time to complete (such as `connect`(2)) was attempted on a nonblocking object (see `ioctl`(2)).

For example, if you issued a `connect`(2) system call on a nonblocking socket, you would normally receive this error. Although the connection is not yet established, the application program does not need to do anything special as the connection is being established asynchronously. (The connection is usually established before you can issue another system call.) If, however, you issued a `write`(2) (or similar) system call before the connection is actually established, the `EWOULDBLOCK` error would be returned.

`130 EALREADY Operation already in progress`
An operation was attempted on a nonblocking object that already had an operation in progress. The application has to wait until the nonblocking operation completes.

`131 ENOTSOCK Socket operation on non-socket`
Certain system calls (for example, `getpeername`(2)) operate only on sockets. Such an operation was attempted on a nonsocket descriptor.

`132 EDESTADDRREQ Destination address required`
A required address was omitted from an operation on a socket. Supply the appropriate address.

`133 EMSGSIZE Message too long`
A message sent on a socket was larger than the internal message buffer. Shorten the message.

`134 EPROTOTYPE Protocol wrong type for socket`
The protocol specified does not support the semantics of the socket type requested. For example, you cannot use the DARPA Internet UDP protocol with type `SOCK_STREAM`. Check to be sure that the operation attempted and the type of socket match.

`135 ENOPROTOOPT Bad protocol option`
A bad option was specified in a `getsockopt` or `setsockopt` system call (see `getsockopt`(2)). Check the various arguments and correct as necessary.

`136 EPROTONOSUPPORT Protocol not supported`
Either the specified protocol has not been configured into the system, or no implementation for it exists. Check the protocol argument on the system call.

`137 ESOCKTNOSUPPORT Socket type not supported`
Either support for the specified socket type has not been configured into the system, or no implementation for it exists. Check the protocol argument on the system call.

`138 EOPNOTSUPP Operation not supported on socket`
For example, trying to use `accept`(2) to accept a connection on a datagram socket (type `SOCK_DGRAM`) is not supported. Check the parameter on the `socket`(2) system call that created the socket.

`139 EPFNOSUPPORT Protocol family not supported`
Either the specified protocol family has not been configured into the system, or no implementation for it exists. Check the protocol argument on the system call.

`140 EAFNOSUPPORT Address family not supported by protocol family`
An address incompatible with the requested protocol was used (for example, you cannot always use PUP Internet addresses with DARPA Internet protocols). For TCP/IP protocols, the address family is `AF_INET`.

`141 EADDRINUSE Address already in use`
Normally, only one socket is allowed to be bound to a local address (Internet address and port number). You sometimes receive this message when you are using the `bind`(2) system call to bind a socket to a local address. Use the `netstat`(1B) command to find out whether the address is already bound.

`142 ENETUNREACH Network is unreachable`
A socket operation was attempted on an unreachable network. Check the destination address. If it is valid, the network is currently unavailable.

`143 ENETRESET Network dropped connection on reset`
The connected host crashed and rebooted. Restart your program.

`144 ECONNABORTED Software caused connection abort`
A connection abort was internal to the local host. Retry the command once. If that fails and the local host is a Cray Research mainframe, see your system support staff.

`145 ECONNRESET Connection reset by peer`
A connection was forcibly closed by a peer. This action normally results when the peer executes a `shutdown`(2) system call.

`146 ENOBUFS No buffer space available`
An operation on a socket or pipe was not performed because the system lacked sufficient buffer space. Retry the operation. Also use the `netstat`(1B) command to see whether you are running out of memory. If you frequently run low on memory, ask your site analyst to configure the system with more mbufs.

`147 EISCONN Socket is already connected`
Either a `connect`(2) request was made on an already connected socket, or a `sendto` or `sendmsg` (see `send`(2)) request on a connected socket specified a destination other than the connected party.

`148 ENOTCONN Socket is not connected`
A request to send or receive data was disallowed because the socket is not connected. Issue a `connect`(2) system call.

`149 ESHUTDOWN Cannot send after socket shutdown`
A request to send data was disallowed because the socket had already been shut down with a previous `shutdown`(2) system call. All `send`(2), `sendto`(2), and `write`(2) system calls fail with this error after a `shutdown`(2) system call has been issued.

`150 ETOOMANYREFS No free TP references available`
A `connect`(2) request or `listen`(2) request failed because no free TP (transport) reference blocks are available. A timer is set whenever a reference block is freed; the reference may not be reused until the timer expires to ensure connection uniqueness within the system. (The time-out period depends on the communication protocol.)

This error may be caused by the maximum number of active connections allowed by the system or by many connections being established and closed in quick succession. In either event, try later at an appropriate time.

`151 ETIMEDOUT Connection timed out`
A `connect`(2) request failed because the connected party did not respond properly within a specified period of time. (The time-out period depends on the communication protocol.) Make sure the remote server process is running. If it is, try later at an appropriate time.

`152 ECONNREFUSED Connection refused`
The connection could not be made, because the target machine actively refused it. You are probably trying to connect to a service that is inactive on the remote host. Make sure that the service is actually running before retrying.

`153 EHOSTDOWN Host is down`
A socket operation failed because the destination host was down. First, recheck the address to be sure it is correct. If it is, retry the operation later when the host is available.

`154 EHOSTUNREACH Host is unreachable`
A socket operation was attempted on an unreachable host. First, recheck the address to be sure it is correct. If it is, retry the operation later when the host is available.

`155 EADDRNOTAVAIL Cannot assign requested address`
This message normally results from an attempt to create a socket with an address not on the local machine. Check the parameters, especially the part number, on the `socket`(2) system call.

`156 ENETDOWN Network is down`
A socket operation encountered a dead network. Retry the operation when the network is available.

UNICOS issues the following BMX-TSS driver interface error codes:

`200 ETPDCNF Tape open rejected due to device configuration`
The device is not configured. You tried to open a tape device, and the system is not configured for tapes, or that particular device is not configured up. Contact your system administrator.

`201 ETPDOPN Tape open rejected, already open to another`
The device is already in use by another process. You tried to issue an `open`(2) or other request to a tape device which is already assigned to another user. Use a different tape device, or contact your system administrator.

`202 ETPDABN Tape I/O request with abnormal status set`
An I/O request completed abnormally due to a previous or current error. Check your `tape.msg` file for more information on the error.

`203 ETPDNRW No write ring on tape device`
The device requires a write ring. This message is obsolete at UNICOS 7.0.

`204 ETPDBDF Bad data returned on tape read`
The `read`(2) function returned incorrect data from the tape. This probably means you have a bad tape.

`205 ETPDEOV Tape end of volume`
This message is obsolete at UNICOS 7.0.

`206 ETPDCLEAR Tape cleared by operator`
The device has been cleared by the operator with a `tpclr`(8) command. Contact your system administrator.

`207 ETPDEOF End-of-file tape mark was read.`
A user tape mark has been read indicating that end-of-file was reached. This is an informative message, and no action is required.

`208 ETPDNODEM Tape daemon is not active.`
The tape daemon has gone down or not been started yet. Contact your system administrator.

`209 ETPDBUFZ User buffer size is not valid.`
You have specified a buffer size that is not valid for either tape list I/O or unbuffered data (using the `-U` option on the `tpmnt`(1) command). The buffer size must be a multiple of 4096 bytes.

- On input requests, the size must exceed the sum of each list entry rounded up to a multiple of 4096 bytes.
- On output requests, the size must exceed the sum of each list entry rounded up to a multiple of 4096 bytes.

Check the buffer size that you are using against what you specified on `tpmnt`.

`210 ETPDRWE A read-after-write or write-after-read occurred.`
Either a read has been requested after a write, or a write has been requested after a read. Both sequences are illegal with tape. Check your program.

`211 ETPDLIST Error in list`
The tape list structure contains one of the following errors:

- You did not specify any entries for tape list I/O.
- You specified an invalid state.
- The byte count is either 0, or it exceeds the maximum block length.

Correct the tape list structure, and then continue processing.

`212 ETPDUERR User error, only close allowed`
Your job can only issue a `close`(2) request due to a previous error. Check your program.

`213 ETPDMBS Maximum tape block size exceeded`
This message is obsolete at UNICOS 7.0.

`214 ETPDLBK Large block tape error`
The block requested is too large for a model E machine; either it is larger than the system maximum, or it is larger than the maximum specified on the `-b` option of your `tpmnt`(1) command. Specify a smaller size block.

`215 ETPDACKERR Acknowledge error before continuing.`
You received a previous error while using asynchronous I/O. If you are executing a Fortran program, the Fortran libraries handle this function. Otherwise, you must acknowledge the error by issuing an `ioctl`(2) system call for `TPC_ACKERR` before any other requests will be honored.

`216 ETPDNOSYSBF No system buffers are available.`
The tape subsystem was unable to obtain the memory needed for the tape subsystem I/O buffers. Contact your system support staff.

`217 ETPDSTOP The IOP is stopped.`
An I/O request was terminated because the tape subsystem is being restarted. Reissue the request later.

`218 ETPDMAXDEVUP Maximum tapes configured up will be exceeded`
A request to configure a device up was terminated because the current number of devices configured up is at the system limit. This limit is defined by the system parameter, `TAPE_MAX_CONF_UP`. Contact your system support staff.

219 `ETPD_PK_BADLEN Packet length is not valid.`
A request to send a packet to an IOP or channel contains a packet length that is not within the valid range, 3 – `EPAK_MAXLEN`. Correct the packet length and reissue the request.

220 `ETPD_PK_NOT_ALLOWED The packet request is not valid.`
An `ioctl`(2) request to send a packet to an IOP or channel device is not valid for that device type. Correct the IOP request packet or reissue the request to the correct device.

221 `ETPD_PK_SEND Error sending packet`
A request could not be sent to an IOP. Contact your system support staff.

222 `ETPD_PK_TIMEDOUT The IOP request timed out.`
The time-out period expired without receiving a response from the IOP. Contact your system support staff.

223 `ETPD_PK_CHAN_UP Channel is configured up`
An attempt to open a channel device failed because the channel is configured up. Diagnostic requests to a channel device can be issued only to a channel that has been configured down by the tape subsystem.

Configure the channel down with the `tpconfig`(8) command and reissue the open request.

224 `ETPD_PK_CHAN_OPENED The channel is already open.`
An attempt to modify the configuration of a channel failed because the channel device is open for diagnostic use. Wait until the channel device is closed and try again.

225 `ETINVCTL The ioctl request is not valid.`
The `ioctl`(2) request issued is not valid. Correct the request and reissue it.

226 `ETPD_BAD_REQT The request issued to the IOP or device is not valid.`
An IOP or device request was terminated either because the contents or the format of the request is incorrect or because the sequence of requests is not valid. Contact your system support staff.

227 `ETPD_BLANK_TAPE A blank tape was detected.`
If this request was issued to an ER90 device, the tape operation was terminated because the command cannot be issued to a device with a blank tape loaded.

If this request was issued to a block multiplexer device, the command was terminated because it cannot execute when the tape is positioned before a blank portion of the tape.

228 `ETPD_NOT_OPER The device is not operational.`
A device request failed because of a hardware error. Additional information can be obtained from the error log. Contact your system support staff.

229 `ETPD_NOT_READY The device is not ready.`
A device request failed because the device is not ready. Switch the device to the ready state and retry the request.

230 `ETPD_EOT End of tape detected`
A device request could not complete because the end of tape was detected. This message is used internally and is not returned to the user.

231 `ETPD_DATA_ERROR  An unrecoverable data error occurred.`
A read or write request failed because of a permanent read or permanent write error. Contact your system support staff.

232 `ETPD_MEDIA Media is not supported`
The configuration of the cassette is not supported.

233 `ETPD_EOR End of recording was detected`
A request failed because the EOR was detected. The EOR is a recorded entity that indicates the end of recording for a partition. Either reposition the tape before the data or record some data at the current position and then reissue the request.

234 `ETPD_LGPS Logical position has not been established`
Your request failed because the logical position had not been established.

If file positioning was turned off using the `-z` option on the `tpmnt`(1) command, you received this message because a request to position to an absolute track address immediately following the tape open was omitted. Position the tape with a `TPC_DMN_REQ` `ioctl` request with a subrequest type of `TR_PABS` and then reissue the request.

If you did not use the `-z` option on the `tpmnt` command, you received this message because a problem has occurred within the tape subsystem. Contact your system support staff.

235 `ETPD_EOM End of media was detected`
The end of the media was detected. Correct the problem and retry. If you receive this message again, contact your system support staff.

236 `ETPD_SYSTEM A tape driver error occurred.`
A tape driver software error occurred. Contact your system support staff.

237 `ETPD_DEVICE  A device error occurred.`
The tape driver received a response that was not valid from the tape device. Contact your system support staff.

238 `ETPD_FORMAT This volume format is not supported.`
Either an ER90 device was unable to format even one partition on a volume, or the format of the volume is not supported. Contact your system support staff.

239 `ETPD_FILETYPE The tape file type is not valid.`
This error is returned to the tape daemon if the current file section is a byte stream file section, but the tape daemon issued a blocked I/O request. This error is not returned to the user.

240 `ETPD_NO_CASSETTE A cassette is not loaded.`
You issued a request that requires a cassette to be loaded to a drive in which a cassette is not currently loaded. Issue a `tpmnt` (1) command and then reissue the original request.

241 `ETPD_TAPE_ADDR The specified tape address is not valid.`
A request to position to an absolute track address failed because the address specified does not exist on the currently mounted cassette. Correct the request and reissue it.

`242 ETPD_DEVDOWN The device must be configured up.`
A request was issued to a downed device, but the request requires that the device be configured up. Use the `tpconfig`(8) command to configure the device up and then reissue the request.

`243 ETPD_NOT_BOF Must be at the beginning of file`
A device request was terminated because the tape is not positioned at the beginning of a file section.

Position the tape with `TPC_DMN_REQ` positioning and then reissue the device request.

`244 ETPD_TAPE_ERROR A tape error occurred.`
A request failed because of a media problem. If an ER90 device was used, the ER90 was unable to locate a byte or block due to a tape fault or to an incorrect tape format. Contact your system support staff.

`245 ETPD_DEV_HUNG Device is hung`
A response was not received from a tape device. Contact your system support staff.

`246 ETPD_MAX_IOREQT Exceeded I/O request size maximum`
You issued an I/O request that exceeds the tape subsystem, IOP, or device limits. Correct the request and then reissue it.

`247 ETPD_ODD_BYTES Cannot issue an odd byte I/O request`
An I/O request was terminated because the previous I/O request output or input an odd number of bytes. An odd byte I/O request is only valid at the end of a file.

Check the requests and correct, as needed. Then reissue the corrected requests.

`248 ETPD_BLKSIZ_DIFF Blocks must be the same size within a file.`
An I/O request was terminated because the previous I/O request output or input a block of a size shorter than the block size defined for the file section. All blocks within a file section must be the same size, excluding the last block in the file section.

Check the requests and correct, as needed. Then reissue the corrected requests.

`249 ETPD_POSACC_ERR Position cannot be accessed`
A request was terminated because it attempted to access data at an odd-byte memory address.

Position the tape with `TPC_DMN_REQ` positioning and then reissue the device request.

UNICOS issues the following communications driver error codes:

`250 ELATE I/O request timeout`
A `read`(2), `reada`(2), `write`(2), `writea`(2), or `listio`(2) request to execute I/O on a communications channel (NSC HYPERchannel, FEI-3, other low-speed device, or HSX) has resulted in no I/O for a certain interval of time. This interval is determined by the type of channel and IOS model.

`251 ENSC NSC HYPERchannel error on write`
No explanation is available for this message.

UNICOS issues the following security violation error codes:

`300 ESYSLV Security level violation`
: The specified security level falls outside the allowed security level range of the process, file system, or UNICOS system. See your security administrator.

`301 EREADV Security read violation`
: An attempt to gain read access to a file has failed because your active security level is less than the file's security level. Raise your active security level to be greater than or equal to the file's security level. If you are not authorized to raise your security level to an appropriate value, see your security administrator.

`302 EWRITV Security write violation`
: An attempt to gain write access to a file has failed because your active security label is not equal to the file's security label. Change your active security label to match the file's security label. If you are not authorized to change your security label to the appropriate value, see your security administrator.

`303 EEXECV Execute security violation`
: An attempt to gain execute/search access to a file has failed because your active security level is less than the file's security level. Raise your active security level to be greater than or equal to the file's security level. If you are not authorized to raise your security level to an appropriate value, see your security administrator.

`304 ECOMPV Security compartment violation`
: An attempt to gain access to a file has failed because your active security compartments do not include all of the file's compartments. For write access, change your active compartments to equal the file's compartments. For read or execute/search access, change your active compartments to include the file's compartments. If you are not authorized to change your active compartments to an appropriate value, see your security administrator.

`305 EMANDV Security mandatory access violation`
: Your active security label does not permit access to a file. For write access, change your active security label to match the file's security label. For read or execute/search access, change your active security label to include the file's security label. If you are not authorized to change your security label to an appropriate value, see your security administrator.

`306 EOWNV Security owner violation`
: You are not authorized to access this file. You must be the file's owner or an appropriate administrator to perform the requested file operation. See your security administrator.

`307 ELEVELV Security level range violation`
: The specified security level falls outside the allowed security level range of the process or file system. See your security administrator.

`308 ESECADM Unauthorized user`
: You are not authorized to make this request.

`309 EFLNEQ Security mount violation`
An attempt to allocate space on a file system has failed because your active security level falls outside the allowed security level range of the file system. Change your active security level to be within the bounds of the file system. If you are not authorized to change your active security level to an appropriate value, see your security administrator.

`310 ENOTEQ Security buffer violation`
This error code is unused.

`311 EPERMIT Security permission violation`
You do not possess the appropriate authorization(s) to perform the requested function. See your security administrator.

`312 EACLV Access list violation`
This error code is unused.

`313 ENOACL No acl list`
This error code is unused.

`314 ESLBUSY Security log in use`
A request to open security log device `/dev/slog` for reading was refused because the device has already been opened for reading by another process. This prohibits two versions of the security log daemon (`slogdemon`) from operating simultaneously. The `slogdemon` should be the only process allowed to request an open on `/dev/slog`. See your security administrator.

`315 ESLNXIO Security log mode violation`
A request to open security log device `/dev/slog` was refused because it attempted the open with write permission. The security log daemon (`slogdemon`) should be the only process to request an open on `/dev/slog`, and then only for reading. See your security administrator.

`316 ESLFAULT Security log read violation`
During the transfer of data from the security log device `/dev/slog` to the disk-resident security log file, a call to a copy out routine returned an error indicating a problem in the memory addressing of the read buffer. See your security administrator.

`317 ESLNOLOG Security log configured SLGOFF`
This error code is unused.

`318 EINTCLSV Security class violation`
The requested integrity class falls outside the allowed integrity class range of the process, or the UNICOS system. See your security administrator. The use of integrity class values is no longer supported.

`319 EINTCATV Security category violation`
The requested category is not included in the allowed categories of the process, or the UNICOS system. See your security administrator.

`320 ENONAL No network authorization list (NAL)`
This error code in unused.

`321 EMNTCMP Security mount compartment violation`
This error code is unused.

`322 EFIFOV Security FIFO violation`
This error code is unused.

`323 EAPPNDV Security append violation`
This error code is unused.

`324 ETFMCATV Security multicategory violation`
This error code is unused.

`325 ECOVERT Covert channel condition`
A file has been created in a wildcard directory by a user who does not own that directory. This is an informative message; you are not required to take corrective action.

`326 ERCLSFY Security label reclassify violation`
This error code is unused.

`327 EPRLABEL Security label printing disabled`
This error code is unused.

`328 ENONSECURE Security is not enabled`
This error code is unused.

`329 ESECFLGV Security flag violation`
A request to set file security flags has failed because the requested flags are not allowed. Specify only security flags that are available on the UNICOS system. See your security administrator.

`330 EHOSTNAL Host not authorized in NAL`
Access to or from an unauthorized host or workstation was attempted. The host is not authorized in the network authorization list (NAL). See your security administrator.

`331 ESLVLNAL Security level outside host’s range`
A security level was detected outside of the security level range authorized for the host in the network authorization list (NAL). The kernel detected this condition when processing the Internet Protocol (IP) security option associated with a datagram. See your security administrator.

`332 ESCMPNAL Security compartment outside host’s range`
A security compartment was detected outside of the security compartment range authorized for the host in the network authorization list (NAL). The kernel detected this condition when processing the Internet Protocol (IP) security option associated with a datagram. See your security administrator.

`333 EMODENAL Illegal transmission for host (NAL)`
An illegal mode (send or receive) of transfer was attempted by a host or workstation. See your security administrator.

`334 ESLVNIF Security level outside network I/F`
A security level was detected outside of the security level range authorized for the UNICOS network interface (I/F). The kernel detected this condition when processing the Internet Protocol (IP) security option associated with a datagram. See your security administrator.

`335 ESCMPNIF Compartment level outside network I/F`
A security compartment was detected outside of the security compartment range authorized for the host in the UNICOS network interface (I/F). The kernel detected this condition when processing the Internet Protocol (IP) security option associated with a datagram. See your security administrator.

`336 ESOCKLVL Security level change of SLS tried`
An illegal attempt was made to change the security level of a single-level socket (SLS) connection. Except for a privilege granted by your security administrator (for example, the network file system (NFS)), all socket connections are created as SLS. See your security administrator.

`337 ESOCKCMP Compartment change of SLS attempted`
An illegal attempt was made to change the security compartment of a single-level socket (SLS) connection. Except for a privilege granted by your security administrator (for example, the network file system (NFS)), all socket connections are created as SLS. See your security administrator.

`338 ENFSAUTH Invalid NFS authentication credential`
The proper authentication credentials were not passed to the network file system (NFS). Check your authentication credentials, or see your security administrator.

`339 ESLVLNRT Security level violation network route`
A security level was detected outside of the security level range authorized for the network route selected, or a route with the correct sensitivity label could not be found. The kernel detected this condition when selecting routes. See your security administrator.

`340 ESCMPNRT Compartments violation for network route`
A security compartment was detected outside of the security compartment authorized for the network route selected, or a route with the correct sensitivity label could not be found. The kernel detected this condition when selecting routes. See your security administrator.

`341 EBADIPSO Bad IP security option`
An illegal IP security option was detected by the kernel. The kernel performs integrity and security checks against each IP security option received. See your security administrator.

`342 ENOIPSO IP security option missing`
An IP security option did not accompany an incoming datagram. The kernel detects this condition by using IP security option information defined in the network authorization list (NAL) for each host/workstation. See your security administrator.

`343 ESLVLMAP Security level mapping error`
A translation error was detected when mapping the security level (between UNICOS form and network form). When necessary, the kernel translates (using the network authorization list (NAL)) the UNICOS security label to the network security label (for outgoing datagrams), and translates the network security label to the UNICOS security label (for incoming datagrams). See your security administrator.

`344 ESCMPMAP Compartment mapping error`
A translation error was detected when mapping the security compartment (between UNICOS form and network form). When necessary, the kernel translates (using the network authorization list (NAL)) the UNICOS security label to the network security label (for outgoing datagrams), and translates the network security label to the UNICOS security label (for incoming datagrams). See your security administrator.

`345 EAUTHFLG Authority flag violation`
An authority protection violation was detected for either an incoming or outgoing datagram with a Basic Security option. See RFC 1108.

`346 EIPSOMAP No map for security option`
No translation table was available to translate the security label for a given host connection. The kernel could access the mapping table identified in the network authorization list (NAL) for the host. See your security administrator.

UNICOS issues the following data migration facility error codes:

`350 EDMRNOLF Not offline file; dmofrq system call`
The file is not an offline file.

`An invalid parameter was specified in the dmofrq`(2) system call.

`352 EDMRWFT Incorrect file type; dmofrq system call`
An incorrect file type was specified with the `m` or `M` option of the `dmofrq`(2) system call.

`353 EDMRNSD Device does not match; dmofrq system call`
The device is incorrect.

`354 EMASS Error occurred in FILESERV storage management system.`
A `FILESERV` error occurred. Contact your system support staff.

`355 EDMOFF Data management system is off`
The data management system is not configured.

`357 EOFFLIN File offline, no automatic retrieval`
The file is offline and automatic retrieval is not selected. Select automatic retrieval by setting `dmmode` to `1`. (See `sh`(1) and `csh`(1).)

`358 EOFLNDD File offline, daemon not available`
The file is offline and the daemon is not available.

`360 EOFLNNR File offline, currently not retrievable`
The file is offline and temporarily cannot be retrieved. You may need to see your system administrator for help.

`361 EOFLRIN File offline, retrieval interrupted`
Retrieval was in process and the user interrupted the process. (The retrieval might continue.)

`362 EOFSPACE File offline, not enough space to retrieve it`
The file is offline and there is not enough space in the file system to retrieve it.

`363 EOFQUOTA File offline, retrieval would exceed disk space quota`
The file is offline, and retrieval is not allowed, because this would exceed the user file quota, the group file quota, or the account file quota.

`365 EDMTRSTD Function reserved for trusted subject only`
The caller issued a `dmofrq`(2) migrate/unmigrate subfunction request, but is not a trusted subject. The data migration daemon is the sole user of this type of request; no user process should ever receive this error.

`367 EDMOFRQ Invalid or inconsistent dmofrq system call parameters`
The caller issued a `dmofrq`(2) system call, but one or more of the parameters are illegal or inconsistent for the requested subfunction. This call is used solely by components of the data migration facility; no user process should ever receive this error.

`368 EDMNBLK Nonblocking recall; data recall from offline media is pending`
An `open` system call (with the `O_NONBLOCK` flag set) failed on a migrated file in which the data is being recalled from an offline media. This indicates that the file is being recalled asynchronously.

`371 ENOIDMAP Named ID map not found`
A reference to an ID map in the kernel failed because the specified map was not found.

`372 EMAPINUSE Named ID map is in use (reference count nonzero)`
An attempt to remove an ID map from the kernel failed because an ID mapping domain is still referencing it.

`373 EMAPTYPE Unknown ID map type`
An attempt to add or delete a map in the kernel failed because the ID map type is not UID or GID.

`374 EDUPMAPNAME Duplicate ID map name`
An attempt to add an ID map to the kernel failed because there is already an ID map with the same name in the kernel.

`375 EGIDMAPSIZE Bad group ID map size`
An attempt to add a GID map to the kernel failed because the size of the GID map is not a multiple of the size of a GID map entry.

`376 EBADDOMAIN ID mapping domain address does not make sense`
An attempt to add or delete an ID mapping domain in the kernel failed because the Internet addresses specified for the mapping domain do not make sense. Either the upper address of the range is numerically less than the lower address, or the address mask is `NULL`.

`377 ENODOMAIN ID mapping domain not found`
An attempt to delete an ID mapping domain from the kernel failed because it was not found.

`378 EBADADDR ID mapping domain overlap error`
An attempt to add an ID mapping domain to the kernel failed because the specified domain overlaps with an existing domain.

`379 ENOUMAPENTRY User ID map entry not found`
An attempt to delete a user's entry from an ID map failed because the entry was not found.

`380 EREMOTEUID Remote hash not found for user ID map entry, ID mapping disabled`
An attempt to delete a user's entry from an ID map failed because after the local UID hash pointer was found and removed, the remote UID hash pointer was not found. The hash table for this map is corrupted.

`381 EUIDTYPE Unknown user ID map entry type`
An attempt to add or delete an entry in a user ID map failed because the type of the entry (which indicates the number of groups in the groups list for this entry) was not valid.

`382 EDUPUMAPENTRY Duplicate user ID map entry`
An attempt to add an entry to a user ID map failed because the entry is already in the ID map.

`383 ESAMEIDMAP This ID mapping domain is unnecessary`
No explanation is available for this message.

`384 EMAPTHRUIDMAP A special ID map is already defined`
No explanation is available for this message.

`385 EMAPTHRUUID Special ID map entry (luid != ruid)`
No explanation is available for this message.

`386 ENOKRBADDR No Kerberos validated address found`
No explanation is available for this message.

`387 EDUPKRBADDR Duplicated Kerberos validated address found`
No explanation is available for this message.

`388 EKRBADDRINUSE Kerberos address reference count nonzero`
No explanation is available for this message.

`393 EIDMADDR Address not found in ID mapping domains`
An NFS request failed because the address of the server is not in the ID mapping domains and ID mapping is enabled.

`394 ESTALE Stale NFS file handle`
An NFS request failed because the information the client has for the remote file system is no longer valid. The NFS file system must be remounted.

`395 ERPCCDRES Cannot decode RPC results`
An NFS request failed because the RPC results cannot be decoded.

`396 ERPCCDARGS Cannot decode RPC arguments`
An NFS request failed because the RPC arguments cannot be decoded.

`397 ERPCCANTSEND Unable to send RPC request`
The RPC request could not be sent.

398 `ERPCAUTH RPC authentication error`
An NFS request failed because an NFS authentication error occurred.

400 `EPKI_NO_PACKETS No packets are available.`
The packet driver does not have any response packets on its packet queue. Retry later.

401 `EPKI_PACKET_LOST A packet was discarded.`
The packet driver discarded a packet received from an IOP. Either the packet interface had not been enabled or there was not an available packet entry on the packet queue when a packet was received.

402 `EPKI_TRUNCATED A packet was truncated.`
This message is obsolete at UNICOS 8.0.

403 `EPKI_TOO_LARGE The packet size exceeds the user buffer size.`
The next packet on the packet driver queue is larger than the amount of memory allocated for the packet in the receive request (`pki_nbytes`).

Allocate a larger block of memory for the packet and reissue the request.

404 `EPKI_ASYNCH_LIM Asynchronous response limit exceeded`
You have issued a request to enable asynchronous responses that attempts to enable more asynchronous responses than allowed by the system limit.

This limit is defined, when the packet driver is started, in the packet driver configuration file, with parameter `MAX_ASYNC.` If this limit is not specified in the configuration file, the asynchronous response limit defaults to 5.

The value may be changed after the packet driver has been started with the `ipi3_option`(8) or `hpi3_option`(8) command.

405 `EPKI_INVAL_CODE The request code is not valid.`
The request packet specified was not valid. Contact your system support staff.

406 `EPKI_NOT_ENABLED The packet interface has not been enabled.`
The packet interface must be enabled before attempting to register a signal (`PKI_SIGNO`), send a packet (`PKI_SEND`), or receive a packet (`PKI_RECEIVE`).

The packet interface is enabled with the following:

```
ioctl, PKI_ENABLE
```

407 `EPKI_REQ_LIM Maximum IOP request limit exceeded`
The number of packets sent by the user, but not yet received, is at the packet driver limit.

- For IOP devices, the request limit is always 1.
- For IPI-3 devices, this limit is defined when the packet driver is started, in the packet driver configuration file, with parameters `MAX_STK_COUNT` and `MAX_NON_CMDLST`. If the limit is not specified in the configuration file, the request limit defaults to 10. This value may be changed after the packet driver has been started with command `ipi3_option`(8) or `hpi3_option`(8).

408 EPKI_BAD_RESYNC The resynchronization code is not valid.
The packet specified in the PKI_SEND request contains a resynchronization code that does not match the resynchronization code of the last command list response received from the IOP.

Correct the PKI_SEND request and reissue it.

409 EPKI_NO_START The IOP driver has not been started.
A packet cannot be sent to an IOP because the IOP driver has not been started.

Start the IOP driver with the ipi3_start(8) or hpi3_start(8) command and then reissue the send request.

410 EPKI_CF_TYPE A configuration type specified is not valid.
A configuration type statement specified in the packet driver configuration file is not valid.

Valid configuration type statements begin with the - character and are followed by one of the following strings: IOPS, CHANNELS, SLAVES, DEVICES, or OPTIONS.

411 EPKI_PARM_ERR The configuration definition specified is not valid.
An error was found in the packet driver configuration file.

412 EPKI_DEV_LIM Exceeded device limits
The requests exceeded the maximum number of IOP devices allowed. The maximum number is limited to MAX_IOPS. Contact your system support staff.

413 EPKI_IOS_ERR An IOS error occurred on an IPI-3 request.
An IOP request did not complete successfully. Contact your system support staff.

414 EPKI_REQT_TYPE The request is not valid for the device type.
The packet specified in a send request (PKI_SEND) is not valid for the device it was issued to. Correct the PKI_SEND request and then reissue it.

415 EPKI_IOCTL_REQT The ioctl request is not valid for the device type.
The ioctl(2) request is not valid for the device it was issued to. Correct the request and then reissue it.

416 EPKI_IOP_SEND Unable to send the request to the IOP
The packet driver was unable to send a packet to an IOP. Contact your system support staff.

417 EPKI_ACTIVE_IOP An IOP is active.
The request could not be processed because an IOP device is open. Wait for all IOP devices to be closed and then reissue the request.

418 EPKI_DEVS_ACTIVE Device(s) on IOP are active
The IOP driver could not be stopped because a device is open.

Wait for the device to be closed and then reissue the request.

419 EPKI_NOT_CONF The driver has not been configured into the system.
IPI-3 packet driver support has not been built into the current system. Contact your system support staff.

`420 EPKI_SYS_ERROR Packet driver error`
An IPI-3 packet driver software error has occurred. Contact your system support staff.

`421 EPKI_NO_DEVICE Requested device not found`
The device specified in the `ioctl`(2) request was not defined in the current configuration. Correct the device specified and then reissue the request.

`422 EPKI_PROC_LIM The process limit per IOP device has been exceeded.`
The open request was rejected because the number of processes with an IOP device open is at the packet driver limit.

When the packet driver is started, this limit is defined in the packet driver configuration file using the `MAX_IOP_PROC` parameter. If the limit is not specified in the configuration file, the IOP device process limit will default to 10.

Wait for a process to close an IOP device and then retry the open request.

The value may be changed after the packet driver has been started with command `ipi3_option`(8) or `hpi3_option`(8).

`423 EPKI_ALREADY_ENBL The packet interface has already been enabled.`
The packet interface has already been enabled. This is an informative message, and no action is required.

`424 EPKI_DEV_CLEAR Device has been cleared`
An `ioctl`(2) request was issued to a device that is in the process of being cleared or has been cleared. After a device has been cleared, no further `ioctl`(2) requests will be accepted until the device has been closed and reopened.

Close the device, reopen it, and then reissue the request.

`425 EPKI_CHAN_DOWN Channel(s) to the device are down`
The packet driver was unable to successfully complete a device clear request because the channel to the device is not in the correct state. Contact your system support staff.

`426 EPKI_HALTIO_ERR Halt I/O request failed`
The packet driver was unable to complete a device clear request because it could not terminate the outstanding IOP activity. Contact your system support staff.

`427 EPKI_SELRST_ERR Selective reset request failed`
The packet driver was unable to complete a device clear request because it could not successfully reset the device. Contact your system support staff.

`428 EPKI_SETATTR_ERR Set attribute request failed`
After a device was cleared, the packet driver was unable to reset the burst size for the device. Contact your system support staff.

429 `EPKI_RESPBUF_LOST The contents of response buffer was lost.`
Because the size of a packet exceeded the IOP maximum packet length, the IOP copied the command portion of the packet into a response buffer allocated by the packet driver.

The contents of this response buffer were lost.

This is an informative message, and no action is required.

430 `EPKI_CMDLIST_LIM The command list limit per IPI-3 has been exceeded.`
The number of command list requests sent by the user, but not yet received, is at the packet driver limit. This limit is defined when the packet driver is started, in the packet driver configuration file with parameter, `MAX_STK_COUNT`.

If the limit is not specified in the configuration file, the limit defaults to 5. This value may be changed after the packet driver has been started with the `ipi3_option`(8) or `hpi3_option`(8) command.

431 `EPKI_DRIVER_DOWN The packet driver is down.`
A packet driver device could not be opened because the packet driver has not been started. Wait until the packet driver has been started and then try again.

432 `EPKI_PEND_SHUTDOWN The packet driver shutdown is pending.`
A packet driver device could not be opened because a shutdown of the packet driver is pending. Wait until the packet driver has been restarted and then try again.

433 `EPKI_DRIVER_UP The packet driver is up.`
The packet driver is up. This is an informative message, and no action is required.

434 `EPKI_PEND_STARTUP The packet driver startup is pending.`
A request to start the packet driver was terminated because another start-up request is pending. This is an informative message, and no action is required.

435 `EPKI_DRIVER_ACTIVE The packet driver is active.`
A packet driver shutdown request failed because the packet driver is active. Wait until the packet driver is inactive and then try again.

436 `EPKI_STOP_DRIVER The request to stop the IOP driver failed.`
The packet driver could not successfully complete a shutdown request because it was unable to stop an IOP driver. Further information can be obtained from the `pki_errno`, `pki_response`, and `pki_extsts` fields in the shutdown response. Contact your system support staff.

437 `EPKI_IOP_NOT_CONF The IOP is not configured.`
An `ioctl`(2) request was issued to an IOP that is not configured, or an `open`(2) request was issued to a device configured on an IOP that has been shut down. Reconfigure the IOP by using the `ipi3_start`(8) command and reissue the request.

438 `EPKI_IOP_SHUTDOWN The IOP has been shut down.`
A request was terminated because the IOP processing the request has been shut down. Reconfigure the IOP by using the `ipi3_start`(8) command and reissue the request.

`439 EPKI_ALREADY_CONFIG A single IOP could not be restarted because it is still configured.`
A configuration request was terminated because the hardware to be defined is already configured.

`500 EFSEMANA (SFS) Fast lock not available`
A request to assign a fast lock to a shared file failed because no more hardware semaphores were available.

`501 EFSNOGROW (SFS) File has allocation restrictions`
A request to change the allocation of a shared file failed because the file was previously set to a state where no allocation changes are allowed.

`502 EFSNOTEXCL (SFS) File is not exclusive`
A nonblocking request to exclusively open a shared file failed because the file is already in an open state by some other process.

`503 EFSEXCLWR (SFS) File is write protected`
A nonblocking request to obtain a read lock on a shared file failed because the file is currently write locked by some other process.

`504 EFSESDOWN (SFS) External semaphore device unavailable`
An attempt to utilize the shared file system has failed because the external semaphore device is unavailable.

`540 ENOTWELLFORMED I/O request not well formed`
An I/O request that is not wellformed has been issued against a file that was opened with the `O_WELLFORMED` option. An I/O request is considered wellformed only if the file offset is exactly on a sector boundary, the I/O request length is exactly a whole number of sectors, and the I/O buffer address in common memory is on a word boundary.

`546 EFSBAD (Panicless File System) File system corrupted`
The panicless file system consistency checking code has detected an error in a file system super block or dynamic block. The file system has been marked in error, and must be repaired with `/etc/fsck`.

`547 EDBAD (Panicless File System) Directory corrupted`
The panicless file system consistency checking code has detected an error in a directory entry. The entry has been marked in error, and must be repaired with `/etc/fsck`.

`549 EIBAD (Panicless File System) File Inode Corrupted`
The panicless file system consistency checking code has detected an error in a file inode table entry. The file has been marked in error, and must be repaired with `/etc/fsck`.

SEE ALSO

`intro`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

Scientific Libraries Reference Manual, Cray Research publication SR–2081

Intrinsic Procedures Reference Manual, Cray Research publication SR–2138

UNICOS Macros and Opdefs Reference Manual, Cray Research publication SR–2403

NAME

`accept` – Accepts a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

`int accept (int` *s*`, struct sockaddr *`*addr*`, int *`*addrlen*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `accept` system call accepts a connection on a socket. It accepts the following arguments:

s Specifies the descriptor for a socket. The socket was created by using `socket`(2), bound to an address with `bind`(2), and is listening for connections after a `listen`(2) request. The `accept` call extracts the first connection on the queue of pending connections, creates a new socket with the same properties as *s*, and allocates a new file descriptor for the socket. If no pending connections exist on the queue, and the socket is not marked as nonblocking, `accept` blocks the caller until a connection exists. If the socket is marked nonblocking, and no pending connections exist on the queue, `accept` returns an error as described in the RETURN VALUES subsection that follows. The accepted socket cannot be used to accept more connections. The original *s* socket remains open and listens for other connections.

addr Specifies the address of a `sockaddr` structure. When a request arrives, the `accept` system call fills this `sockaddr` structure with the address of the client that placed the request. The domain in which the communication is occurring determines the exact format of the *addr* argument.

addrlen Specifies the address of an integer. The `accept` system call fills this integer with the length of the address that was placed in the `sockaddr` structure pointed to by *addr*. Initially, it must contain the amount of space to which *addr* points; on return, it contains the actual number of bytes in the address that is returned.

The `accept` call sets up send and receive socket buffers (sockbufs) using the sockbuf space limit of the listening *s* socket. The `accept` call fails and returns an `ELIMIT` error if this call would cause the user's per-session sockbuf space limit to be exceeded. The original *s* socket remains open and continues to listen.

This call is used with connection-based socket types; it is currently used with `SOCK_STREAM`.

To determine whether a connection is ready to be accepted, instead of issuing the `accept` system call, you can issue the `select`(2) system call and set the bit for the socket file descriptor in the read mask (*readfds*).

For protocols that require an explicit confirmation, the `accept` call merely dequeues the next connection request; it does not imply confirmation. Confirmation can be implied by a standard read or write operation on the new file descriptor; rejection can be implied by closing the new socket.

You can obtain user connection request information without confirming the connection by issuing a `recvmsg`(2) call with a `msg_iovlen` value of 0 and a nonzero `msg_control` value, or by issuing a `getsockopt`(2) call. Similarly, you can provide information about user connection rejection by issuing a `sendmsg`(2) call and providing only the control information, or by issuing a `setsockopt`(2) call.

NOTES

If *addrlen* is less than the size of the address of the connecting entity (that is, less than the size of a `struct sockaddr`), the `accept` call truncates its result to fit into the available space.

If the `SOCKET_MAC` configuration option is enabled, the active security label of the process must be greater than or equal to the security label of the socket. The `SOCKET_MAC` configuration option is part of the TCP/IP configurable feature variables list in `uts/cf/Nmakefile`. For more information, see the `connect`(2) man page.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_MAC_READ`	The process is allowed to override the security level and compartment restrictions when the `SOCKET_MAC` configuration option is enabled.

RETURN VALUES

If `accept` completes successfully, it returns a nonnegative integer that is a descriptor for the accepted socket; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `accept` call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	If the `SOCKET_MAC` configuration option is enabled, the process does not meet the security level and compartment requirements and does not have the appropriate privilege.
`EBADF`	Descriptor is invalid.
`EFAULT`	Argument *addr* or argument *addrlen* is not in the user address space in which it can be written.
`EINVAL`	Socket is not bound or socket is not listening.
`ELIMIT`	The user's socket buffer space limit is exceeded.
`ENOTSOCK`	Descriptor is not a socket.

`EWOULDBLOCK` Socket is marked nonblocking, and no connections are present to be accepted.

EXAMPLES

This server program shows how to use the `accept` system call in context with other TCP/IP calls. (Some system calls in this example are not supported on Cray MPP systems.) The program simply creates a TCP/IP socket, waits for a client process from some host to attempt a connection, accepts the connection, and forks a child process to provide the service to the client.

The original (parent) server loops back to look for additional connection attempts while the temporary (child) server reads a string of data sent by the client process.

```
/*  Server side of client-server socket example. For client side,
    see socket(2).
    Syntax: server portnumber & */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

main(int argc, char *argv[])
{
     int s, ns;
     struct sockaddr_in src;          /* source socket address */
     int len=sizeof(src);
     char buf[256];

     /* create port */
     src.sin_family = AF_INET;
     src.sin_port = atoi(argv[1]);
     src.sin_addr.s_addr = 0;

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("server, unable to open socket");
          exit(1);
     }

     while (bind(s, (struct sockaddr *) &src, sizeof(src)) < 0) {
          printf("Server waiting on bind...\n");
          sleep(1);
     }

     listen(s, 5);
```

```
          while (1) {
               ns = accept(s, (struct sockaddr *) &src, &len);
               if (ns < 0) {
                    perror("server, accept failed");
                    exit(1);
               }

               if (fork() == 0) {
                    /* in child server */
                    close(s);     /* child will use socket ns, parent uses s */
                    read(ns, &buf, sizeof(buf));
                    printf("Server read: %s\n", buf);
                    close(ns);
                    exit(0);
               }
               close(ns);         /* close socket used by child */
          }
     }
```

FILES

`/etc/config/spnet.conf`	Contains the network access list
`/usr/adm/sl/slogfile`	Receives security log records
`/usr/include/sys/socket.h`	Contains definitions related to sockets, types, address families, and options
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`bind`(2), `connect`(2), `getsockopt`(2), `listen`(2), `select`(2), `sendmsg`(2), `setsockopt`(2), `socket`(2)

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

NAME

`access` – Determines accessibility of a file

SYNOPSIS

```
#include <unistd.h>
int access (const char *path, int amode);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `access` system call determines the accessibility of the path name pointed to by the *path* argument. It checks for the file access permissions indicated by *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

It accepts the following arguments:

path Points to a file path name.

amode Identifies the bit pattern to check against the file's bit pattern denoting access permission. Construct the bit pattern in *amode*, as follows:

`00` or `F_OK`	Check existence of file
`01` or `X_OK`	Execute (search)
`02` or `W_OK`	Write
`04` or `R_OK`	Read
`020` or `G_OK`	Check for `set-gid` bit
`040` or `U_OK`	Check for `set-uid` bit
`0400` or `EUID_OK`	Test using the effective IDs rather than the real IDs

The owner of a file has permission checked with respect to the owner read, write, and execute mode bits; members of the file's group other than the owner have permissions checked with respect to the group mode bits; and all others have permissions checked with respect to the other mode bits.

NOTES

If the file has an access control list, users that are not the file owner have permissions checked with respect to the access control list. Users who are not affected by entries in the access control list have permissions checked with respect to the other mode bits.

Permission to the file is also based on a comparison between the active security label of the process and the security label of the file. These comparisons are summarized as follows:

- For read, execute, or search permission, the active security label of the process must dominate the security label of the file.
- For write permission, the active security label of the process must equal the security label of the file.

If the file is a labeled device, the active security label of the process must fall within the security label range of the device.

Only appropriately authorized users are granted permission to files that are in the `OFF` state or that are multilevel.

The process must be granted search permission to every component of the path prefix via the permission bits and access control list.

The process must be granted search permission to every component of the path prefix via the security label.

If `FSETID_RESTRICT` is enabled, only a process with appropriate privileges can be granted write permission to set-user-ID or set-group-ID files.

A process with the effective privileges shown are granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted read, execute, search, or write permission to the file via the permission bits and access control list.
`PRIV_FSETID`	If `FSETID_RESTRICT` is enabled, the process is granted write permission to the set-user-ID or set-group-ID file.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_READ`	The process is granted read, execute, or search permission to the file via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted read, execute, search, or write permission to the file. The super user or a process with the `suidgid` permission can override the restriction introduced by the `FSETID_RESTRICT` system configuration option.

RETURN VALUES

If `access` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `access` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Permission bits of the file mode do not permit the requested access.
EACCES	Search permission is denied on a component of the path prefix.
EACCES	The file is in the `OFF` state and the calling process does not have appropriate privileges.
EACCES	The file is a multilevel file and the calling process does not have appropriate privileges.
EACCES	The security label of the file does not allow the requested access.
EACCES	If the `FSETID_RESTRICT` and `PRIV_SU` configuration options are enabled, the process does not have appropriate privilege to gain write permission to the set-user-ID or set-group-ID file.
EFAULT	The *path* argument points outside the allocated process address space.
EMANDV	The security label range of a device does not allow the requested access.
ENAMETOOLONG	The *path* argument is longer than `PATH_MAX` characters.
ENOENT	The specified file does not exist.
ENOENT	Read, write, or execute (search) permission is requested for a null path name.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	Write access is requested for a file on a read-only file system.
ETXTBSY	Text file is busy.

Mandatory access violations are recorded in the security log for these conditions:

Error Code	Description
EACCES	The user's security label does not allow access to the file.
EINTCATV	The user's active category does not match the file's category.
EMANDV	The caller is attempting to open an existing, labeled device but does not have an active security label that falls within the authorized security label range of the device.

FORTRAN EXTENSIONS

The `access` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER amode, ACCESS, I
I = ACCESS (path, amode)
```

path may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFACCESS`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example illustrates how to use the `access` system call to check on the existence of a file before issuing an `open`(2) call. The `access` request verifies that file `datafile` exists before an attempt is made to open it.

```
int fd;

if (access("datafile", F_OK)) {
     fprintf(stderr, "File datafile does not exist.\n");
}
else {
     fd = open("datafile", O_RDONLY);
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `access` system call

SEE ALSO

`chmod`(2), `stat`(2)

`PXFACCESS`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`acct` – Enables or disables process accounting

SYNOPSIS

`#include <unistd.h>`

`int acct (char *`*path*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `acct` system call enables or disables the system process accounting routine. If the routine is enabled, an accounting record is written to an accounting file for each process that terminates. An `exit`(2) call or a signal can cause termination. Only a process with appropriate privilege can use this system call.

The `acct` system call accepts the following argument:

path Points to a path name that specifies the accounting file. The accounting file format is given in `acct`(5).

If *path* is nonzero and no errors occur during the system call, the accounting routine is enabled. If *path* is 0 and no errors occur during the system call, it is disabled.

If the accounting routine is already enabled and *path* differs from the accounting file currently in use, the accounting file will be switched to *path* without the loss of any accounting information.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_ACCT`	The process is allowed to use this system call.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_ACCT` permbit is allowed to use this system call.

RETURN VALUES

If `acct` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `acct` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`EACCES`	The file specified by *path* is not a regular file.
`EACCES`	Write permission is denied for the specified accounting file.
`EFAULT`	The *path* argument points to an illegal address.
`EISDIR`	The specified file is a directory.
`ENOENT`	One or more components of the accounting file path name do not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have appropriate privilege to use this system call.
`EROFS`	The specified file resides on a read-only file system.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `acct` system call

SEE ALSO

`exit`(2)

`acct`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`acctctl` – Checks status of, enables, and disables process, daemon, and record accounting

SYNOPSIS

```
#include <sys/types.h>
#include <sys/accthdr.h>
#include <sys/acct.h>

int acctctl (int fnc, void *act);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `acctctl` system call checks the status of, enables, and disables process, daemon, and record accounting. It accepts the following arguments:

fnc Identifies a function to be performed as follows:

Function	**Description**
`AC_DMDAUTHORIZED`	Returns an indication of whether the executing user is authorized to enable/disable accounting.
`AC_DMDSTART`	Enables the indicated accounting method.
`AC_DMDSTOP`	Disables the indicated accounting method.
`AC_DMDCHECK`	Checks the current state of a specific accounting method.
`AC_DMDSTAT`	Checks the current state of all accounting methods.

act Points to either an `actctl`, `actstat`, or `actstat` structure, depending on the function (i.e. *fnc*) to be performed.

This parameter is ignored for the `AC_DMDAUTHORIZED` function.

Enabling an accounting method requires an `actctl` structure which defines the daemon/record accounting identifier to be enabled (`actctl.ac_stat.ac_id`), the name of the file to write accounting data to (`actctl.ac_path`), and an optional parameter, which is defined by the accounting method being enabled (`actctl.ac_stat.ac_param`).

Disabling an accounting method requires an `actctl` structure which defines the daemon/record accounting identifier to be disabled (`actctl.ac_stat.ac_id`).

Checking the state of an accounting method requires an `actstt` structure which defines the daemon/record accounting identifier being checked (`actstt.ac_id`). On a successful return, the accounting method's current state (i.e. `ACS_ON`, or `ACS_OFF`) and optional parameter's value (`actstt.ac_stt` and `actstt.ac_param`, respectively) are set.

Checking the state of all accounting methods requires an `actstt` structure which defines the number of accounting status entries available (`ac_stat.ac_sttnum`). On a successful return, each available entry is filled in with the current state of an accounting method up to the number of accounting methods defined in the kernel. An accounting method's status entry can be accessed by using its defined identifier as the index into the `ac_stat`[] array returned.

NOTES

Only a process with appropriate privilege can use this system call to enable and/or disable accounting. However, no special privilege is required to check/status accounting states.

When enabling accounting, if the type of accounting specified is already enabled, the accounting file being used for data collection will be closed and switched to the file specified without losing any accounting information.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the file, the active security label of the process must equal the security label of the file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ACCT`	The process is allowed to use this system call to enable/disable accounting.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix through the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the file through the permission bits and access control list.
`PRIV_MAC_READ`	The calling process is granted search permission to every component of the path prefix through the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the file through the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the file. The super user is allowed to use this system call to enable/disable accounting.

RETURN VALUES

If `acctctl` completes successfully, a value of 0 is returned and the structure pointed to by *act* is filled in as indicated above. If `acctctl` completes unsuccessfully, a value of −1 is returned, the structure pointed to be *act* is not modified and `errno` is set to indicate the error.

For the `AC_DMDAUTHORIZED` function, a value of 0 is returned if the executing user is authorized to enable/disable accounting. A value of −1 is returned and `errno` is set to `EPERM`, if the executing user is not authorized to enable/disable accounting.

ERRORS

The `acctctl` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied on a component of the path prefix.
EACCES	The process is not granted write permission to the file.
EACCES	The file specified by *path* is not an ordinary file.
EFAULT	The *act* argument points to an illegal address.
EINVAL	An invalid argument was specified.
EISDIR	The file specified is a directory.
ENOENT	A component of the path prefix or the file does not exist.
EPERM	The process does not have appropriate privilege to use this system call.
EROFS	The specified file resides on a read-only file system.

FILES

/usr/include/acct/dacct.h	Defines daemon accounting files
/usr/include/sys/acct.h	Defines the `actctl`, `actstt`, and `ac_stat` structures
/usr/include/sys/accthdr.h	Defines daemon/record identifiers

SEE ALSO

`acct`(2), `dacct`(2)

`acct`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`acct`(8), `csaswitch`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`acctid` – Changes account ID of a process

SYNOPSIS

```
#include <unistd.h>
int acctid (int pid, int acid);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `acctid` system call changes the account ID of the specified process. Only a process with appropriate privilege can set the account ID.

The `acctid` system call accepts the following arguments:

pid Specifies the *pid* of the target process. A *pid* of 0 means the current process.

acid Specifies the value of the new account ID. The value must be either nonnegative or –1. An *acid* of –1 means no change.

For more information about changing the account ID of a user, see `newacct`(1).

NOTES

The active security label of the calling process must be greater than or equal to the active security label of the specified process.

To set the account ID of a process, the active security label of the calling process must be equal to the active security label of the specified process.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ACCT`	The calling process is allowed to set the account ID.
`PRIV_MAC_READ`	The calling process is allowed to override the restriction that its active security label must be greater than or equal to the security label of the specified process.
`PRIV_MAC_WRITE`	The calling process is allowed to override the security label restriction when setting the account ID of a process.
`PRIV_POWNER`	The calling process is considered the owner of the specified process.

If the `PRIV_SU` configuration option is enabled, the super user is considered the owner of the specified process and is allowed to set the account ID. The super user is allowed to override all security label restrictions.

RETURN VALUES

If `acctid` completes successfully, it returns the previous account ID of the specified process; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `acctid` system call fails if one of the following error conditions occurs:

Error Code	Description
EINVAL	One of the arguments contains an invalid value.
EPERM	The calling process does not have appropriate privilege to set the account ID.
EPERM	The calling process does not have appropriate privilege to use this system call.
ESRCH	The specified process could not be found.
ESRCH	The caller does not own the specified process and does not have appropriate privilege.
ESRCH	The calling process does not meet the security label requirements and does not have appropriate privilege.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `acctid` system call

SEE ALSO

`newacct`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`adjtime` – Corrects the time to allow synchronization of the system clock

SYNOPSIS

```
#include <sys/time.h>
int adjtime (struct timeval *delta, struct timeval *olddelta);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `adjtime` system call adjusts the system's notion of the current time, as returned by `gettimeofday`(2). It advances or retards it by the amount of time specified in the `struct timeval` (defined in header file `sys/time.h`) pointed to by *delta*.

The `adjtime` system call accepts the following arguments:

delta Points to a `timeval` structure.

olddelta Points to a structure that contains, upon return, the time still to be corrected from the earlier call. If *olddelta* is a null pointer, the corresponding information is not returned.

The adjustment is effected by speeding up (if that amount of time is positive) or slowing down (if that amount of time is negative) the system clock by some small percentage, generally a fraction of 1%. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to `adjtime` may not be finished when `adjtime` is called again.

This call is used in time servers that synchronize the clocks of computers in a local area network. Such time servers slow down the clocks of some machines and speed up the clocks of others to bring them to the notion of network time.

Only a process with appropriate privilege can use this system call.

The adjustment value is silently rounded to the resolution of the system clock.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_TIME`	The process is allowed to use this system call.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call.

RETURN VALUES

If `adjtime` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `adjtime` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	*olddelta* points to a region of the process' allocated address space that is not writable.
`EPERM`	The process does not have appropriate privilege to use this system call.

SEE ALSO

`gettimeofday`(2)

`date`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

alarm, _lwp_alarm – Sets a process alarm clock

SYNOPSIS

```
#include <unistd.h>
unsigned int alarm (unsigned int sec);
unsigned int _lwp_alarm (unsigned int sec);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The alarm system call instructs the alarm clock of the calling process to send the SIGALRM signal to the calling process after a specified number of real-time seconds has elapsed; see signal(2).

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

The alarm and _lwp_alarm system calls accept the following argument:

sec Specifies the number of real-time seconds. To cancel a previous alarm request, set *sec* to 0.

On Cray MPP systems, the alarm system call sets an alarm only for the processing element (PE) on which it is called. It has no effect on any other PE of the application.

The _lwp_alarm system call ensures that the alarm signal is sent to the specific member of the multitasking group that called _lwp_alarm. In contrast, alarm results in an alarm signal being sent to an arbitrary thread.

NOTES

The _lwp_alarm system call provides compatibility with the behavior of alarm previous to UNICOS 9.0. It is a transitional tool since it may disappear in a future release of the UNICOS operating system. This compatibility issue affects only multitasked applications.

RETURN VALUES

The alarm system call returns the amount of time previously remaining in the alarm clock of the calling process.

FORTRAN EXTENSIONS

The `alarm` system call can be called from Fortran as a function:

```
INTEGER sec, ALARM, I
I = ALARM (sec)
```

Alternatively, `alarm` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER sec
CALL ALARM (sec)
```

The Fortran program must not specify both the subroutine call and the function reference to `alarm` from the same procedure.

EXAMPLES

This example shows how to use the `alarm` request to notify the invoking process when a specific amount of time has expired. After the specified time (10 seconds), the `SIGALRM` signal is sent to the process, interrupting the process.

Using the `signal`(2) system call, the program requests that the function `handlr` be entered when the `SIGALRM` signal is received; otherwise, the process will usually terminate upon the receipt of the signal. After the function `handlr` executes, control is returned to the point of interruption.

```
#include <signal.h>
#include <unistd.h>

main()
{
     void handlr(int signo);

     signal(SIGALRM, handlr);
     alarm(10);

     /* After executing 10 seconds, SIGALRM signal interrupts program. */
     /* Execution resumes here after processing SIGALRM signal. */
}

void handlr(int signo)
{
     signal(signo, handlr);
     /* Process SIGALRM signal here and then return. */
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `alarm` system call

SEE ALSO

`pause`(2), `sigctl`(2), `signal`(2), `sigset`(2)

NAME

`bind` – Binds a name to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int bind (int s, struct sockaddr *name, int namelen);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `bind` system call assigns a name to an unnamed socket. It accepts the following arguments:

s	Specifies the descriptor of the socket to be bound.
name	Points to the address of the `sockaddr` structure that contains the local address to which the socket should be bound.
namelen	Specifies length of the address, pointed to by *name*. The length is measured in bytes.

When a socket is created by using `socket`(2), it is assigned a descriptor *s* and exists in a name space (address family), but it has no name assigned. The `bind` call requests that the name *name* be assigned to the socket.

The rules used in name binding vary among communication domains.

NOTES

The active security label of the process must equal the security label of the socket.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_WRITE`	The process is allowed to override the security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security label restrictions.

RETURN VALUES

If `bind` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `bind` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	The process does not meet the security label requirements, and does not have appropriate privilege.
`EADDRINUSE`	Specified address is already in use.
`EADDRNOTAVAIL`	Specified address is not available from the local machine.
`EBADF`	Descriptor *s* is not valid.
`EFAULT`	Argument *name* is not in a valid part of the user address space.
`EINVAL`	Socket is already bound to an address or *namelen* is not the size of a valid address for the specified address family.
`ENOTSOCK`	Descriptor *s* is not a socket.

See `open`(2) for file system-related errors.

EXAMPLES

This server program shows how to use the `bind` system call in context with other TCP/IP calls. (Some system calls in this example are not supported on Cray MPP systems.) The program simply creates a TCP/IP socket, waits for a client process from some host to attempt a connection, accepts the connection, and forks a child process to provide the requested service to the client.

The original (parent) server loops back to look for additional connection attempts while the temporary (child) server reads a string of data that the client process sends.

```
/*  Server side of client-server socket example. For client side,
    see socket(2).
    Syntax: server portnumber & */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

main(int argc, char *argv[])
{
     int s, ns;
     struct sockaddr_in src;          /* source socket address */
     int len=sizeof(src);
     char buf[256];

     /* create port */
     src.sin_family = AF_INET;
     src.sin_port = atoi(argv[1]);
     src.sin_addr.s_addr = 0;

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("server, unable to open socket");
          exit(1);
     }

     while (bind(s, (struct sockaddr *) &src, sizeof(src)) < 0) {
          printf("Server waiting on bind...\n");
          sleep(1);
     }
```

```
          listen(s, 5);

          while (1) {
               ns = accept(s, (struct sockaddr *) &src, &len);
               if (ns < 0) {
                    perror("server, accept failed");
                    exit(1);
               }

               if (fork() == 0) {
                    /* in child server */
                    close(s);     /* child will use socket ns, parent uses s */
                    read(ns, &buf, sizeof(buf));
                    printf("Server read: %s\n", buf);
                    close(ns);
                    exit(0);
               }
               close(ns);         /* close socket used by child */
          }
     }
```

FILES

`/usr/include/sys/socket.h`	Header file for sockets
`/usr/include/sys/types.h`	Header file for types

SEE ALSO

`connect`(2), `getsockname`(2), `listen`(2), `open`(2), `socket`(2), `unlink`(2)

`inet`(4P), `intro`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`brk`, `sbrk`, `sbreak` – Changes data segment space allocation

SYNOPSIS

```
#include <unistd.h>
int brk (char *endds);
char *sbrk (int incr);
long *sbreak (int incr);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `brk`, `sbrk`, and `sbreak` system calls dynamically change the amount of space allocated for the data segment of the calling process; see `exec`(2). The change is made by resetting the break value of the process and allocating the appropriate space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is initialized to 0.

The `brk`, `sbrk`, and `sbreak` system calls accept the following arguments:

endds Specifies break value to be set (`brk` only). To specify the return value of the `ulimit`(2) system call as the value for *endds*, you must convert the number returned by `ulimit`(2) to a character pointer, as follows:

```
brk(((char *)0) + ulimit(3,0));
```

incr Specifies the number of bytes (`sbrk`) or words (`sbreak`) to add to the break value. To decrease the allocated space, specify *incr* as a negative number.

CAUTIONS

The use of the `brk`, `sbrk`, or `sbreak` system call directly interferes with the processing of library routine `malloc`(3C), which the calling sequence uses to implement run-time stack space.

RETURN VALUES

If `brk` completes successfully, a value of 0 is returned; `sbreak` and `sbrk` return the old break value. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `brk`, `sbrk`, or `sbreak` system call fails without making any change in the allocated space if one of the following error condition occurs:

Error Code	Description
`ENOMEM`	More space is specified in the argument than is allowed by a system-imposed maximum (see `limit`(2) and `ulimit`(2)).
`EMEMLIM`	More memory space was requested than is allowed for the processes attached to this lnode. The maximum value is set by the `-c` option of the `shradmin`(8) command. This error appears only on systems running the fair-share scheduler.

FORTRAN EXTENSIONS

The `brk` system call can be called from Fortran as a function:

```
INTEGER endds, BRK, I
I = BRK (endds)
```

Alternatively, `brk` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER endds
CALL BRK (endds)
```

The Fortran program must not specify both the subroutine call and the function reference to `brk` from the same procedure.

The `sbrk` system call can be called from Fortran as a function:

```
INTEGER incr, SBRK, I
I = SBRK (incr)
```

The `sbreak` system call can be called from Fortran as a function:

```
INTEGER incr, SBREAK, I
I = SBREAK (incr)
```

EXAMPLES

The following examples illustrate how to use the `brk`, `sbrk`, and `sbreak` system calls to expand and decrease the size of the calling process. The first three examples, involving the same expansion task, highlight differences in these calls.

Example 1: This `brk` request expands the size of the calling process by 1000 octal words (8 bytes per word).

A `sbrk` request first determines the current break value for the process from which the new break value is calculated. Then, `brk` expands the calling process size.

```
char *brkp;
int rtrn;

brkp = sbrk(0);      /* return current break value for process. */
rtrn = brk(brkp + (8 * 01000));
```

Example 2: The `sbrk` request expands the size of the calling process by 1000 octal words (8 bytes per word). The return value placed in `ptr` points to the beginning of the newly allocated block of 1000 octal words.

```
char *ptr;

ptr = sbrk(8 * 01000);
```

Example 3: The `sbreak` request expands the size of the calling process by 1000 octal words (8 bytes per word). The return value placed in `ptr` points to the beginning of the newly allocated block of 1000 octal words.

```
long *ptr;

ptr = sbreak(01000);
```

Example 4: This `sbreak` request decreases the size of the calling process by 2000 octal words (8 bytes per word):

```
long *ptr;

ptr = sbreak(-02000);
```

Example 5: The following `brk` system call increases the size of the calling process to the maximum allowed for the user's process:

```
brk(((char *)0) + ulimit(3,0));
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `brk`, `sbrk`, and `sbreak` system calls

SEE ALSO

`exec`(2), `limit`(2), `ulimit`(2)

`malloc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`shradmin`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`chacid` – Changes disk file account ID

SYNOPSIS

```
#include <unistd.h>
int chacid (char *path, int acid);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `chacid` system call changes the account ID associated with a disk file. (The file can be the original file or a symbolic link.) The `chacid` system call accepts the following arguments:

path Specifies the path name of the file to be changed.

acid Specifies the account ID or –1. If *acid* is –1, the current account ID is returned, and no change is made.

All users may call `chacid` with an *acid* of –1; however, only a process with appropriate privilege may call `chacid` with an account ID other than –1.

NOTES

The active security label of the calling process must be greater than or equal to the active security label of the file.

To set the account ID of a file, the active security label of the calling process must equal the active security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ACCT`	The calling process is allowed to set the account ID of the file.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path via the permission bits and access control list.
`PRIV_MAC_READ`	The calling process is allowed to override the restriction that its active security label must be greater than or equal to the security label of the file.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path via the security label.

`PRIV_MAC_WRITE` The calling process is allowed to override the security label restriction when setting a file's account ID.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix. The super user is allowed to set the account ID of a file. If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security label restrictions.

RETURN VALUES

If `chacid` completes successfully, it returns the previous account ID associated with the file; otherwise, a value of –1 is returned, and `errno` is set to indicate the error. If the *acid* argument is –1, the return value is the current account ID associated with the file.

ERRORS

The `chacid` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`EINVAL`	The file is not on a local file system.
`EINVAL`	The calling process does not meet security label requirements and does not have appropriate privilege.
`EINVAL`	The *acid* argument contains an invalid value.
`EPERM`	The calling process does not have appropriate privilege to set the file account ID.
`EQACT`	A file or inode quota limit was reached for the current account ID.

FILES

`/usr/include/unistd.h` Contains C prototype for the `chacid` system call

SEE ALSO

`acctid`(2)

`chacid`(1), `newacct`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`chdir` – Changes working directory

SYNOPSIS

```
#include <unistd.h>
int chdir (const char *path);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `chdir` system call causes a specified directory to become the current working directory; that is, the starting point for path searches for path names not beginning with `/`. The `chdir` system call accepts the following argument:

path Points to the directory path name.

NOTES

To be granted search permission to a component of the path name, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path name.

RETURN VALUES

If `chdir` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `chdir` system call fails and the current working directory remains unchanged if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Search permission is denied for any component of the path name.
`EFAULT`	The *path* argument points outside the allocated process address space.
`ENOENT`	The specified directory does not exist.
`ENOTDIR`	A component of the path name is not a directory.

FORTRAN EXTENSIONS

The `chdir` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER CHDIR, I
I = CHDIR (path)
```

Alternatively, `chdir` can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
CALL CHDIR (path)
```

The Fortran program cannot specify both the subroutine call and the function reference to `chdir` from the same procedure. *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFCHDIR`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

The following `chdir` request changes the current working directory in the invoking process environment to the parent directory of the current working directory:

```
if (chdir("..")) {
     fprintf(stderr, "The directory change was unsuccessful.\n");
     exit(1);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `chdir` system call

SEE ALSO

`chroot`(2)

`PXFCHDIR`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`chdiri` – Changes a directory by using the inode number

SYNOPSIS

`int chdiri (long` *dev*`, long` *ino*`, long` *gen*`);`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `chdiri` system call provides the user with a pathless change-directory operation on native UNICOS file systems. It locates a directory by using the inode number, and then it causes this directory to become the current directory.

The `chdiri` system call accepts the following arguments:

dev Specifies the device number. This number is built by the `makedev` macro that is defined outside of the kernel.

ino Specifies an inode number for the directory as reported by the `ls –i` command.

gen Specifies the generation number of the inode.

This provides a unique identification for a specific file. The generation number changes when an inode is reused. To print inode generation values, use the `fck`(1) command with the `i` and `l` options.

NOTES

Only a process with appropriate privilege can use this system call.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call.

A process with the `PRIV_MAC_READ` and `PRIV_DAC_OVERRIDE` effective privileges is allowed to use this system call. See the effective privilege discussion in the NOTES section of the `chdir`(2) man page for additional privilege requirements. The `chdir`(2) search access discussions do not apply to this system call.

RETURN VALUES

If `chdiri` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `chdiri` system call fails and the current working directory remains unchanged if one of the error conditions listed on the `chdir`(2) man page occurs.

FILES

/usr/include/sys/sysmacros.h Contains a description of the makedev macro

SEE ALSO

chdir(2)

fck(1), ls(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

General UNICOS System Administration, Cray Research publication SG–2301

NAME

chkpnt – Checkpoints a process, multitask group, or job

SYNOPSIS

```
#include <sys/category.h>
#include <sys/restart.h>
int chkpnt (int category, int id, char *path, long flags);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The chkpnt system call creates a file containing all the information needed to restore the target processes identified by *category* and *id* to their saved execution state by the restart(2) system call. The file created is referred to as a *restart file*.

The chkpnt system call accepts the following arguments:

category Specifies C_PROC for a process or C_SESS for a job (or interactive session).

id Specifies the *pid* or *jid* corresponding to *category*. *id* = 0 assumes current process or current job, respectively.

path Specifies the path name of the restart file to be created.

flags Identifies optional actions.

The flags present in this field are OR'ed together to define the optional action to be performed by chkpnt. Currently, the only defined flag value is CHKPNT_KILL, which causes the target processes to die after the recovery image is complete.

By default, the restart file is protected from user modification and can be read only by the owner (file mode 0400) unless one of the following conditions is true:

- One of the processes in the chkpnt collection has an effective user ID (UID) different from the effective UID of the process performing chkpnt.
- One of the processes in the chkpnt collection has a security label that is different from the security label of the process performing chkpnt.
- One of the processes in the chkpnt collection has one of the following flags set:
  - PC_NOCORE (The process does not have read access to its executable image.)
  - PC_SECCORE (The process is permitted to use privilege and the SECURE_MAC configuration option is enabled.)
- One of the processes in the chkpnt collection is a setuid application.

The restart file is recognized by the system as a restart file because the type field of the restart file inode identifies the file as a regular file and the `S_IRESTART` (restart file attribute) bit is also set (see `/usr/include/sys/stat.h`).

Whenever any process is selected to be included in a restart file, all of its multitask group sibling processes are also included, because the meaningful recovery of any process requires that all of its multitask group siblings also be restored on recovery.

Processes with open pipes can be checkpointed and restarted if their pipe connections do not go outside the job or multitask group being checkpointed. To checkpoint a process with open pipes, all of its pipe connections must terminate with processes that are also to be included in the restart file.

Processes with open files that reside on network file system (NFS) file systems can be checkpointed and restarted. To restart a process with open NFS files, the NFS file systems on which the files reside have to be mounted unless the NFS file systems are managed by the automounter. In this case, the automounter will try to remount the file systems automatically.

Processes with open files that reside on Distributed File System (DFS) file systems can be checkpointed and restarted. The following conditions must exist in order for a user to restart a process with an open DFS file.

- The DFS client must be running on the local host.
- The DFS server must be running on the host where the file resides.

Access to DFS files is controlled by the user's Distributed Computing Environment (DCE) credentials as opposed to user identification (UID) and group identification (GID). DFS credentials consist of Kerberos tickets stored in a special file. When a process is checkpointed, a reference to these credentials is stored in the restart file. The credentials must still be present and valid when `restart`(2) is performed. If the credentials are no longer present or have expired, accesses to DFS files that are performed after the `restart`(2) system call will appear to be from the UID –2.

If the `chkpnt` system call writes the restart file into a directory that is being accessed via DFS, that directory must reside on a Cray Research DFS server. DFS servers of other manufacturers do not support the restart file type; if `chkpnt` tries to write a restart file into one of these directories, the call fails without returning an error.

Processes with unlinked files can be checkpointed and restarted if the total size of all unlinked files in use by the target process set is within the size limit established by the system administrator. See the `MAX_UNLINKED_BYTES` system variable in the `/usr/src/uts/c1/cf/config.h` file to see the site local definition.

With SSD solid-state storage devices, processes that are using secondary data segments (SDS) can be checkpointed and restarted if sufficient disk space is available to contain an image of the process SDS area within the restart file. An `ENOSDS` error may occur at restart time if the SDS area available at that time is less than what was in use at checkpoint time. The `ENOSDS` error means that `restart`(2) must be retried at a later time when sufficient SDS space is available.

Processes using online tape files cannot be checkpointed or restarted.

NOTES

The following restrictions apply to processes and jobs (including interactive sessions) that are to be checkpointed:

- Only a process with appropriate privilege may checkpoint or restart another user's job or process.
- The active security label of the job of the calling process must dominate the security label of the job or process being checkpointed, unless the caller has appropriate privilege.
- Processes with open pipes may be checkpointed and restarted successfully if the following two conditions are met:
  - All openings of the pipe file must be contained within the process collection being checkpointed.
  - All I/O operations on the pipe must be atomic with respect to the `chkpnt` system call. This condition is a limit on the size of an I/O operation: either `PIPE_BUF` bytes, or (`v_maxpipe * 4096`) bytes. `PIPE_BUF` is found in the `sys/param.h` file. `v_maxpipe` is a member of the `var` structure in the `sys/var.h` file.
- All files that a process was using when it was checkpointed must be present when the process is restarted. These files include all open files, any shared-text executables that the process was using such as shells, and the present working directory. In the restart file, each of these files is identified by its inode number and the minor number of the file system. If either changes, the `restart`(2) system call fails, and the call returns an `EFILERM` error. For example, if a file system is restored by `/etc/restore`, any process that was using files on that file system and that was checkpointed before the restore, will fail to restart. After the restore, each file on the file system has a different inode number than it did when the process was checkpointed.
- Processes using online tapes cannot be checkpointed or restarted.
- Processes using shared memory segments (CRAY T90 series systems only) cannot be checkpointed or restarted.

A process with the effective privilege shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the directory containing the new restart file via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_WRITE`	The active security label of the calling process is considered equal to the active security label of every process being checkpointed.

`PRIV_MAC_WRITE`	The process is granted write permission to the directory containing the new restart file via the security label.
`PRIV_POWNER`	The calling process is considered the owner of every process being checkpointed.

If the `PRIV_SU` configuration option is enabled, the super user is considered the owner of every process being checkpointed. The super user is granted all access necessary to create the new restart file.

RETURN VALUES

If `chkpnt` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `chkpnt` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Search permission is denied on a component of the restart file path prefix.
`EACCES`	The directory in which the restart file is to be created does not permit writing.
`EAGAIN`	One of the processes being checkpointed was never in a state that allowed checkpointing during the last 120 seconds. The `chkpnt` system call may be attempted again.
`EEXIST`	A file by the name of *path* already exists.
`EEXIST`	An open NFS file descriptor that is unlinked cannot be saved in the restart file.
`EFAULT`	The *path* argument points outside the allocated process address space.
`EFBIG`	To complete the checkpoint operation, the restart file would have to be larger than the file size limit of the calling process or the maximum file size.
`EFILERM`	One or more of the target processes has one or more unlinked files open, and the sum of the sizes of all unlinked files open by the target processes exceeds the site-configured, unlinked file, contents recovery limit.
`EFILESH`	One or more of the target processes has an open, unlinked regular file that is also open by one or more processes outside the target process set.
`EFOREIGNFS`	An operation that is supported only on local file systems was attempted on a nonlocal (foreign) file system.
`EINVAL`	An invalid argument was passed to the system call.
`EINVAL`	The caller has specified that a restart file of an entire job be created, and one or more of the processes in the job is in a process group whose process group leader is outside the job.
`ENFILE`	The system inode table is full.

`ENOENT`	A component of the restart file path prefix does not exist.
`ENOENT`	The restart file path name is null.
`ENOSPC`	Insufficient file space is available to create the restart file.
`ENOTBLK`	An unrecoverable resource is associated with the target process set.
`ENOTDIR`	A component of the restart file path prefix is not a directory.
`ENOTTY`	One or more of the processes has an unrecoverable character device open.
`ENOTTY`	The caller has specified that a restart file of an entire job be created, and two or more processes in the job have different controlling ttys.
`EPERM`	The caller was not running as root and specified a *pid* for which the caller's real or effective *uid* differed from the real or effective user ID of the target process.
`EPERM`	The caller did not have appropriate privilege and specified *category* as `C_SESS`.
`EPERM`	The active security label of the caller did not dominate that of the job or process being checkpointed.
`EPIPE`	One or more of the target processes has an open pipe that goes outside the target process set.
`EQACT`	A file or inode quota limit was reached for the current account ID.
`EQGRP`	A file or inode quota limit was reached for the current user ID.
`EQUSR`	A file or inode quota limit was reached for the current group ID.
`EROFS`	The *category* argument is `C_PROC` and no process exists with the requested process ID.
`ESHMA`	The process has a shared memory segment or segments attached (CRAY T90 series systems only), and cannot be checkpointed.
`ESOCKTNOSUPPORT`	Either support for the specified socket type has not been configured into the system, or no implementation for it exists. Check the protocol argument on the system call.
`ESRCH`	The specified restart file would reside on a read-only file system.
`EXDEV`	The specified restart file would reside on a foreign file system (for example, a remote file accessed with NFS).
`EXDEV`	One or more of the target processes has a file open on a foreign file system (for example, a remote file accessed with NFS).
`EXDEV`	One or more of the target processes has another process open through the */proc* file system and that process is not included in the target process set. A file or inode quota limit was reached for the current group ID.

EXAMPLES

The following examples illustrate different uses of the `chkpnt` system call.

Example 1: The `chkpnt` system call produces a checkpoint file (named `chkpnt.pid`) of the invoking process in the current working directory:

```
char filename[256], pid_char[8];
int pid;

pid = getpid();               /* get pid of current process */
sprintf(pid_char, "%d", pid);/* convert pid to char format */
strcpy(filename, "chkpnt."); /* create filename for chkpnt with */
strcat(filename, pid_char);  /* format chkpnt.pid */

if (chkpnt(C_PROC, pid, filename, 0) != 0) {
     perror("chkpnt failed");
}
```

Example 2: The `chkpnt` system call produces a checkpoint file (named `chkpnt.jid`) of the job containing the invoking process in the current working directory. After the checkpoint file is created, the job is immediately terminated.

```
char filename[256], jid_char[8];
struct jtab jdata;
int jid;

jid = getjtab(&jdata);       /* get jid of current job */
sprintf(jid_char, "%d", jid);/* convert jid to char format */
strcpy(filename, "chkpnt."); /* create filename for chkpnt with */
strcat(filename, jid_char);  /* format chkpnt.jid */

if (chkpnt(C_SESS, jid, filename, CHKPNT_KILL) != 0) {
     perror("chkpnt failed");
}
```

SEE ALSO

`chmod`(2), `chown`(2), `creat`(2), `getpid`(2), `mknod`(2), `open`(2), `pipe`(2), `restart`(2), `setuid`(2)

`chkpnt_util`(1), `chkptint`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

chmem – Retrieves or modifies system physical memory availability

SYNOPSIS

```
#include <unistd.h>
#include <sys/map.h>

int chmem (long request, long *result);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The chmem system call provides a mechanism for determining the physical memory available to the host system and, for appropriately privileged processes, the capability to modify how much physical memory is available.

The chmem system call has the following arguments:

request Specifies the amount (in words) by which the system's notion of physical memory will be changed. All *requests* are rounded up to the nearest 512-word size. To retrieve the current notion of system physical memory, specify 0 as the value of *request*.

result Specifies an address.

The chmem system call returns the system's current notion of physical memory after the requested change has been considered, at the address specified by *result*. If *result* is null, no data is returned.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_RESOURCE	The process is allowed to modify system physical memory availability.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_SYSPARAM permbit is allowed to modify system physical memory availability.

RETURN VALUES

If chmem completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The chmem system call fails if one of the following error conditions occurs:

Error Code	Description
EAGAIN	An attempt to reduce the system's notion of physical memory could not be satisfied, probably because an unmovable process was locked into the portion of physical memory being taken down. Retry the procedure later.
EINVAL	The *result* address supplied was invalid.
ENOSPC	An attempt to increase the system's notion of physical memory would expand beyond the compile-time configured maximum amount of memory.
ENXIO	When the call was manipulating a bit map of memory, an error occurred. The chmem interface was disabled, returning −1 to all subsequent *requests*.
EPERM	A nonzero *request* was made by a process without appropriate privilege.

FILES

/usr/include/sys/types.h	Contains types required by ANSI X3J11
/usr/include/unistd.h	Contains C prototype for the chmem system call

NAME

`chmod`, `fchmod` – Changes the mode of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod (const char *path, mode_t mode);

int fchmod (int fildes, mode_t mode);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `chmod`)

DESCRIPTION

The `chmod` and `fchmod` system calls set the access permission portion of the specified file's mode as specified by the following arguments:

path Points to a file path name.

mode Specifies the bit pattern denoting the file's access permission. (See the header file, `sys/stat.h`, for a description of these bits.)

fildes Specifies the file descriptor.

To set the mode of a file, the process must be the file owner or have appropriate privilege. To set the mode of a restart file, the process must have appropriate privilege.

If the process is not a member of the file's owning group and the process does not have appropriate privilege, then the file `S_ISGID` mode bit (set group ID on execution) is cleared.

If the `S_ISGID` bit (set group ID on execution) is set and the `S_IXGRP` bit (execute or search by group) is not set, mandatory file or record locking will exist on a regular file. This can affect subsequent calls to `creat`(2), `listio`(2), `open`(2), `read`(2), `reada`(2), `trunc`(2), `write`(2), and `writea`(2) on this file.

If the `S_ISVTX` (sticky) bit is set on a directory, only the directory owner or a process with appropriate privilege can delete or rename files in that directory.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

The process must be granted write permission to the file via the security label. That is, the active security label of the process must equal the security label of the file.

If the FSETID_RESTRICT system configuration option is enabled, only a process with appropriate privilege can set the set-user-ID or set-group-ID mode bits. If a process does not have appropriate privilege, the set-user-ID and set-group-ID mode bits are cleared.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
PRIV_FOWNER	The process is considered the file owner.
PRIV_FSETID	If the FSETID_RESTRICT system configuration option is enabled, the process is allowed to set the set-user-ID mode bits.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_WRITE	The calling process is granted write permission to the file via the security label.
PRIV_RESTART	The process is allowed to set the mode of a restart file.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix. The super user is allowed to set the mode of a restart file. The super user is considered the file owner.

If the PRIV_SU configuration option is enabled, the super user is allowed to set the set-user-ID and set-user-ID mode bits and is granted write permission to the file via the security label.

RETURN VALUES

If chmod or fchmod completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The chmod or fchmod system call fails and the file mode remains unchanged if one of the following error conditions occurs:

Error Code	**Description**
EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *path* argument points outside the allocated process address space.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The specified file resides on a read-only file system.

`EMANDV`	User’s compartments and level are not equal to that of the file.
`EMANDV`	The calling process does not have MAC write access to the file to which the file descriptor refers.
`EMANDV`	The process is not granted write permission to the file via the security label.
`EPERM`	The process is not the file owner and does not have appropriate privilege.
`EPERM`	The file is a restart file, and the process does not have appropriate privilege.
`EPERMIT`	User does not have permission to set or change the mode of a file to setuid or setgid.
`EPERMIT`	If the `FSETID_RESTRICT` system configuration is enabled, the process does not have appropriate privilege to set the set-user-ID or set-group-ID mode bits.

FORTRAN EXTENSIONS

The `chmod` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER CHMOD, mode, I
I = CHMOD (path, mode)
```

Alternatively, `chmod` can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
INTEGER mode
CALL CHMOD (path, mode)
```

The Fortran program cannot specify both the subroutine call and the function reference to `chmod` from the same procedure. *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFCHMOD`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

The following examples illustrate different uses of the `chmod` system call.

Example 1: The `chmod` system call grants read/write permission to the owner of `file1` and only read permission to all other users:

```
if (chmod("file1", 0644) == -1) {
     perror("chmod failed");
}
```

Example 2: Setting a file's setgid bit (02000) and clearing the group execute permission bit enables mandatory file locking for the file. This `chmod` call establishes mandatory file locking for the `datafile` file in addition to granting the other file access permissions.

```
if (chmod("datafile", 02644) == -1) {
     perror("chmod setting mandatory locking failed");
}
```

SEE ALSO

`chkpnt`(2), `chown`(2), `creat`(2), `fcntl`(2), `listio`(2), `mknod`(2), `open`(2), `read`(2), `reada`(2), `stat`(2), `trunc`(2), `write`(2), `writea`(2)

`PXFCHMOD`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`chown`, `lchown`, `fchown` – Changes owner and group of a file

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int chown (const char *path, uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
int fchown (int fildes, uid_t owner, gid_t group);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `chown`, `lchown`, and `fchown` system calls assign a new owner and group to a file. These system calls accept the following arguments:

path Points to a file path name.

owner Specifies the numeric value of the new owner ID.

group Specifies the numeric value of the new group ID.

fildes Specifies the file descriptor.

If the `POSIX_CHOWN_RESTRICTED` option is enabled, only a process with appropriate privilege may change file ownership.

If the process does not have appropriate privilege, then the file `S_ISUID` (set-user-ID) and `S_ISGID` (set-group-ID) mode bits are cleared.

If *path* is a symbolic link, `chown` will change the owner and group of the file referenced by the symbolic link. `lchown` will change the owner and group of the symbolic link itself.

If *owner* or *group* is specified as –1, the corresponding ID of the file is not changed.

Only the owner of a file or a process with appropriate privilege may change file ownership.

Only a process with appropriate privilege can change the owner of a restart file.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

The process must be granted write permission to the file via the security level and compartments. That is, the active security label of the process must equal the security label of the file.

If the FSETID_RESTRICT configuration option is enabled, only a process with appropriate privilege is allowed to change the owner of set-user-ID or set-group-ID files.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_CHOWN	If the POSIX_CHOWN_RESTRICTED option is enabled, the process is allowed to change file ownership.
PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
PRIV_FOWNER	The process is considered the file owner.
PRIV_FSETID	The process is allowed to change the owner of a set-user-ID or set-groups-ID file.
PRIV_FSETID	The process is allowed to preserve the set-user-ID or set-groups-ID mode bits.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_WRITE	The calling process is granted write permission to the file via the security label.
PRIV_RESTART	The process is allowed to set the mode of a restart file.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix. The super user is considered the file owner and is allowed to change the owner of a restart file. The super user is allowed to preserve the set-user-ID and set-groups-ID mode bits. If the POSIX_CHOWN_RESTRICTED option is enabled, the super user or a process with the PERMBITS_CHOWN permbit is allowed to change file ownership.

If the PRIV_SU configuration option is enabled, the super user is granted write permission to the file via the security label. If the PRIV_SU and FSETID_RESTRICTED configuration options are enabled, the super user is allowed to change the owner of a set-user-ID and set-groups-ID file.

RETURN VALUES

When chown, lchown, or fchown completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The `chown`, `lchown`, or `fchown` system call fails and the owner and group of the specified file remains unchanged if one of the following error conditions occurs:

Error Code	**Description**
EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *path* argument points outside the allocated address space of the process.
EMANDV	User's security label is not equal to that of the file.
EMANDV	The calling process does not have MAC write access to the file to which the file descriptor refers.
EMANDV	The process is not granted write permission to the file via the security label.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The process is not a file owner and does not have appropriate privilege.
EPERM	The file is a restart file and the process does not have appropriate privilege.
EPERMIT	User is a trusted user, but does not have `suidgid` permission.
EPERMIT	If the `FSETID_RESTRICT` configuration option is enabled, the process does not have appropriate privilege to change the owner of a set-user-ID and set-group-ID file.
EQGRP	A file or inode quota limit was reached for the new group ID.
EQUSR	A file or inode quota limit was reached for the new user ID.
EROFS	The specified file resides on a read-only file system.

FORTRAN EXTENSIONS

The `chown` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER CHOWN owner, group, I
I = CHOWN (path, owner, group)
```

Alternatively, `chown` can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
INTEGER owner, group
CALL CHOWN (path, owner, group)
```

The Fortran program cannot specify both the subroutine call and the function reference to `chown` from the same procedure. *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFCHOWN`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example shows how the `chown` request changes ownership on a file (`chown` is a restricted operation for most users).

The `getpwnam` (see `getpwent`(3C)) library routine first locates the user and group IDs for user `joe` from the `/etc/passwd` file. `chown` changes the ownership of file `myfile` to user `joe`.

```
#include <pwd.h>
#include <unistd.h>

main()
{
     struct passwd *pwptr;

     pwptr = getpwnam("joe");
     if (chown("myfile", pwptr->pw_uid, pwptr->pw_gid) == -1) {
          perror("chown failed");
     }
}
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `chown`, `fchown`, and `lchown` system calls

SEE ALSO

`chkpnt`(2), `chmod`(2)

`getpwnam`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`PXFCHOWN`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`chroot` – Changes the root directory

SYNOPSIS

```
#include <unistd.h>
int chroot (const char *path);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `chroot` system call causes the specified directory to become the root directory; that is, the starting point for searches for path names beginning with a slash (`/`). It accepts the following argument:

path Points to a directory path name. The `chroot` system call does not affect your working directory.

The process must have appropriate privilege to use this system call. For more information on permission bits (permbits), see *General UNICOS System Administration*, Cray Research publication SG–2301.

The `..` entry in the root directory is interpreted to mean the root directory itself. Thus, you cannot use `..` to access files outside the subtree rooted at the root directory.

NOTES

To be granted search permission to a component of the path, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to use this system call.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path via the security label.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_CHROOT` permbit is allowed to use this system call. The super user is granted search permission to every component of the path.

RETURN VALUES

When `chroot` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `chroot` system call fails and the root directory remains unchanged if one of the following error conditions occurs:

Error Code	Description
`EACCES`	The process is denied search permission to a component of the specified path.
`EFAULT`	The *path* argument points outside the allocated address space of the process.
`ENOENT`	The specified directory does not exist.
`ENOTDIR`	Any component of the path name is not a directory.
`EPERM`	The process does not have appropriate privilege to use this system call.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `chroot` system call

SEE ALSO

`chdir`(2)

`udbgen`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`close` – Closes a file descriptor

SYNOPSIS

```
#include <unistd.h>
int close (int fildes);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `close` system call closes the specified file descriptor. It accepts the following argument:

fildes Specifies a file descriptor. It is obtained from an `accept`(2), `creat`(2), `dup`(2), `fcntl`(2), `open`(2), `pipe`(2), `socket`(2), or `socketpair`(2) system call.

RETURN VALUES

If `close` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `close` system call fails if one of the following error condition occurs:

Error Code	**Description**
`EBADF`	The *fildes* argument is not a valid open file descriptor.
`EINTR`	The `close` system call was interrupted.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `close` system call

SEE ALSO

`accept`(2), `creat`(2), `dup`(2), `exec`(2), `fcntl`(2), `open`(2), `pipe`(2), `shutdown`(2), `socket`(2), `socketpair`(2)

NAME

`cmptext` – Compares the supplied character sequence with the privilege text of the calling process

SYNOPSIS

```
#include <sys/priv.h>
#include <sys/tfm.h>

int cmptext (long *seq, long flags);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `cmptext` system call compares the supplied character sequence with the privilege text of the calling process.

The `cmptext` system call accepts the following arguments:

seq Specifies the character sequence. The character sequence is a long integer but can be passed as a character sequence enclosed by single quotation marks (for example, `cmptext('`*charseq*`',...)`.

flags Specifies a bit mask that can contain any combination of the `ROOT_EFFECTIVE` and `ROOT_REAL` flags. The flags are defined as follows:

Flag	**Description**
`ROOT_EFFECTIVE`	Indicates that any process whose effective user ID is 0 automatically has the supplied privilege text when `PRIV_SU` is enabled.
`ROOT_REAL`	Indicates that any process whose real user ID is 0 automatically has the supplied privilege text when `PRIV_SU` is enabled.

A *flags* value that includes neither `ROOT_EFFECTIVE` or `ROOT_REAL` indicates that user ID 0 does not automatically have the specified privilege text. A *seq* value of 0 causes `cmptext` to return successfully if the calling process has null privilege text.

RETURN VALUES

A return value of 0 indicates that the privilege text of the calling process is identical to the supplied character sequence, or that the caller meets the user ID 0 requirements as specified previously.

A positive return value indicates that the supplied character sequence does not match the privilege text of the process and does not meet the user ID 0 requirements specified previously.

A return value of –1 indicates that an error has occurred, and an error code is stored in `errno`.

ERRORS

The `cmptext` system call fails if the following error condition occurs:

Error Code	Description
`EINVAL`	A value supplied for *seq* or *flags* was not valid.

NAME

`connect` – Initiates a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (int s, struct sockaddr *name, int namelen);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `connect` system call initiates a connection on a socket. It accepts the following arguments:

`s` Specifies the descriptor of the socket to connect. When the socket is of the `SOCK_RAW` or `SOCK_DGRAM` type, the `connect` system call permanently specifies the peer to which datagrams are sent. If the socket is of the `SOCK_STREAM` type, this call tries to connect to another socket.

name Points to a `sockaddr` structure that contains the destination address of the socket to which the *s* socket is to be connected. This destination address is in the address domain of the socket.

namelen Specifies the length of the destination address. The length is measured in bytes.

Each address domain uniquely interprets the *name* argument. Generally, stream sockets can successfully connect only once; datagram sockets can use the `connect` call multiple times to change association. Datagram connections can dissolve an association by connecting to an invalid address such as a null address.

NOTES

If *namelen* is less than the size of the address of the connecting entity (that is, less than the size of a `struct sockaddr`), the `accept`(2) system call truncates its result to fit into the available space.

The `connect` system call is subjected to additional security rules. The two sockets being connected each have security attributes that are inherited from their associated processes. These attributes must be equal if the `SOCKET_MAC` option is enabled. In addition, the network and remote host have security-attribute ranges, which are specified in the network access list (NAL) portion of the `spnet.conf` configuration file and administered by using the `spnet`(8) command.

If the `SOCKET_MAC` configuration option is not enabled, the security attributes of the socket are not required to be equal, but the security range of the process, which is specified in the UDB for the user, must include the minimum label for the remote host as specified in the NAL. `SOCKET_MAC` is part of TCP/IP configurable feature variables list in `uts/cf/Nmakefile`.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_WRITE	The process is allowed to override the security label restrictions when the SOCKET_MAC option is enabled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions when the SOCKET_MAC option is enabled.

RETURN VALUES

If connect completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The connect system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	If the SOCKET_MAC configuration option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EADDRINUSE	Address is already in use.
EADDRNOTAVAIL	Specified address is unavailable on this machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EBADF	Descriptor *s* is invalid.
ECONNREFUSED	Attempt to connect is forcefully rejected.
EFAULT	Argument *name* specifies an area outside the process address space.
EISCONN	Socket is already connected.
ENETUNREACH	Network cannot be reached from this host.
ENOTSOCK	Descriptor *s* is not a socket.
ETIMEDOUT	Connection establishment timed out without establishing a connection.
EWOULDBLOCK	Socket is nonblocking; the connection cannot be completed immediately. You can use the select(2) call to select the socket while it is connecting by selecting it for writing.

EXAMPLES

This client program shows how to use the connect system call in context with other TCP/IP calls. The program creates a TCP/IP socket and then attempts to establish a connection between the newly created socket and the socket within the server program on the designated server host. If a connection is successful, the client process sends a string of data to the server process.

```
/*  Client side of client/server socket example. For server side,
    see socket(2).
    Syntax: client hostname portnumber */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

/*      in in.h is this socket structure
 *
 *       Socket address, internet style.
 *
 *       struct sockaddr_in {
 *          short   sin_family;
 *          u_short sin_port;
 *          struct  in_addr sin_addr;
 *          char    sin_zero[8];
 *      };
 */

#define DATA "Test message from client to server."

main(int argc, char *argv[])
{
     int s;
     struct sockaddr_in dest;          /* destination socket address */
     struct hostent *hp;               /* host structure pointer */

/* Converts host name into network address. */
     hp = gethostbyname(argv[1]);

     dest.sin_family = hp->h_addrtype; /* addr type (AF_INET) */
     bcopy(hp->h_addr_list[0], &dest.sin_addr, hp->h_length);
     dest.sin_port = atoi(argv[2]);

     /* create port */

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("client, cannot open socket");
          exit(1);
     }
     if (connect (s, (struct sockaddr *) &dest, sizeof(dest)) < 0) {
```

```
                close(s);
                perror("client, connect failed");
                exit(1);
           }
           write(s, DATA, sizeof(DATA));

           close(s);
           exit(0);
      }
```

FILES

`/etc/config/spnet.conf`	Network access list file
`/usr/adm/sl/slogfile`	Receives security log records
`/usr/include/sys/socket.h`	Contains definitions related to sockets, types, address families, and options
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`accept`(2), `getsockname`(2), `select`(2), `socket`(2)

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

NAME

`cpselect` – Selects which processors may run the process

SYNOPSIS

```
#include <unistd.h>
int cpselect (int pid, int mask);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `cpselect` system call specifies the physical processors that may execute a process as specified by the following arguments: It accepts the following arguments:

pid Specifies the process ID of process to execute. A *pid* of 0 means the current process.

mask Specifies the bit mask indicating physical processors. Processor A or 0 is 01, processor B or 1 is 02, and so on. A *mask* of 0 signifies the use of any available CPU; a *mask* of –1 does not change the select mask, but it returns the previous mask. The CPU mask is inherited by child processes.

NOTES

The *mask* argument is silently limited to the available processors. If all processors chosen are unavailable, the process is allowed to run on any processor.

The active security label of the calling process must be greater than or equal to the security label of the affected process.

To set the processor execution mask of a process, the active security label of the calling process must equal the security label of the affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_READ`	The active security label of the calling process is considered greater than or equal to the security label of the affected process.
`PRIV_MAC_WRITE`	The active security label of the calling process is considered equal to the security label of the affected process.
`PRIV_POWNER`	The calling process is considered the owner of the affected process.

If the `PRIV_SU` configuration option is enabled, the super user is considered to be the owner of the affected process. If the `PRIV_SU` configuration option is enabled, the super user is allowed to bypass security label restrictions.

RETURN VALUES

If `cpselect` completes successfully, it returns the previous *mask* value; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `cpselect` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EINVAL`	The *pid* argument contains an invalid value.
`EPERM`	The real or effective user ID of the calling process does not match the real or effective user ID of the affected process, and the calling process does not have appropriate privilege.
`ESRCH`	No process can be found to match the *pid*.

FORTRAN EXTENSIONS

The `cpselect` system call can be called from Fortran as a function:

```
INTEGER pid, mask, CPSELECT, I
I = CPSELECT (pid, mask)
```

Alternatively, `cpselect` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER pid, mask
CALL CPSELECT (pid, mask)
```

The Fortran program cannot specify both the subroutine call and the function reference to `cpselect` from the same procedure.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `cpselect` system call

NAME

`creat` – Creates a new file or rewrites an existing one

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat (const char *path, mode_t mode);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `creat` system call creates a new ordinary file or prepares to rewrite an existing file. The call is equivalent to an `open`(2) system call of the following form:

```
open (path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

For a description of the arguments used in this call, see the `open`(2) man page.

The `creat` system call accepts the following arguments:

path Points to a file to be created or an existing file to be rewritten.

mode Specifies the bit pattern denoting the file's access permission. (See `stat`(2) for the description of these bits.)

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID of the process, and the group ID is set to the group ID of the directory in which the file is created. The low-order 12 bits of the file mode are set to the value of *mode* modified as follows: all bits set in the process' file mode creation mask are cleared. See `umask`(2).

If `creat` completes successfully, the file descriptor is returned, and the file is opened for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across `exec`(2) system calls (see `fcntl`(2)). No process can have more than `OPEN_MAX` files open simultaneously. You can create a new file with a mode that forbids writing.

NOTES

The active security label of the calling process must fall within the security label range of the file system on which the new file will reside.

If the `FSETID_RESTRICT` option is enabled, only a process with appropriate privilege can create set-user-ID or set-group-ID files.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the parent directory, the active security label of the process must equal the security label of the directory.

To be granted write permission to an existing file, the active security label of the process must equal the security label of the file.

A process with the effective privilege shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the parent directory or to an existing file via the permission bits and access control list.
`PRIV_FSETID`	When the `FSETID_RESTRICT` option is enabled, the process is allowed to create set-user-ID or set-group-ID files.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the parent directory or to an existing file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the parent directory and to an existing file. When `FSETID_RESTRICT` is enabled, the super user is allowed to create set-user-ID and set-group-ID files.

RETURN VALUES

If `creat` completes successfully, a nonnegative integer, (the file descriptor) is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `creat` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	The file exists and write permission is denied.

EACCES	The file does not exist, and the directory in which the file is to be created does not permit writing.
EACCES	Search permission is denied on a component of the path prefix.
EAGAIN	The file exists, mandatory file and record locking is set, and outstanding record locks exist on the file (see chmod(2)).
EFAULT	The *path* argument points outside the allocated address space of the process.
EFLNEQ	The active security label of the calling process does not fall within the range of the file system on which the new or rewritten file will reside.
EISDIR	The specified file is an existing directory.
EMFILE	OPEN_MAX file descriptors are currently open.
ENFILE	The system file table is full.
ENOENT	A component of the path prefix does not exist.
ENOENT	The path name is null.
ENOTDIR	A component of the path prefix is not a directory.
EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.
EROFS	The specified file resides or would reside on a read-only file system.
ETXTBSY	The text file is busy.
EWRITV	If the FSETID_RESTRICT option is enabled, the process does not have appropriate privilege to create a set-user-ID or set-group-ID file.

FORTRAN EXTENSIONS

The creat system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER  mode, CREAT, I
I =  CREAT (path, mode)
```

Alternatively, creat can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
INTEGER mode
CALL CREAT (path, mode)
```

The Fortran program must not specify both the subroutine call and the function reference to `creat` from the same procedure. *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFCREAT`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

SEE ALSO

`chmod`(2), `close`(2), `dup`(2), `exec`(2), `fcntl`(2), `lseek`(2), `open`(2), `read`(2), `stat`(2), `umask`(2), `write`(2)

`PXFCREAT`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`cutimes` – Updates user execution time

SYNOPSIS

```
#include <sys/types.h>
#include <sys/utimes.h>

struct utms *cutimes (struct utms *mytimes);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `cutimes` system call lets users have a structure in user memory continually updated with user execution time information. From this `utms` structure, users can determine how much execution time has accumulated in an interval without calling the operating system.

The `cutimes` system call accepts the following argument:

mytimes Specifies the address of the structure to receive the data.

If the address is 0, the structure is no longer updated.

The `utms` structure contains the following members:

```
time_t   utms_update;  /* RT clock at start of this connect */
time_t   utms_utime;   /* Total user time during previous connects */
```

Since the `utms` structure is only updated at the time of connection to a process, the current user time accumulated since the process began can be calculated as follows:

```
time = mytimes.utms_utime + (rtclock() - mytimes.utms_update);
```

This method will yield the desired results most of the time. But there is a small chance that the operating system will change the `utms` values in the middle of the calculation. A guaranteed method is shown in the EXAMPLES section and is also implemented in the `cpused`(3C) function.

Update of the `utms` structure stops if one of the following occurs:

- A memory contraction places the structure outside user memory.
- An `exec`(2) system call is executed.

NOTES

The times in the `utms` structure are figured from the time the process began execution, not from the invocation of the `cutimes` call. Monitoring the `utms` structure at the user level is somewhat tricky because the user may be interrupted when looking at the structure. The `cpused`(3C) routine monitors this structure.

RETURN VALUES

If `cutimes` completes successfully, the address of the user structure is returned, with 0 meaning that the feature is disabled. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `cutimes` system call fails if the following error condition occurs:

Error Code	**Description**
EFAULT	The *mytimes* argument is out of the user's memory space.

EXAMPLES

This example shows how to use the `cutimes` system call to compute the amount of user execution time for a section of code within a user's program. The amount of user time is computed in hardware clock ticks and seconds.

```
#include <sys/types.h>
#include <sys/utimes.h>
#include <time.h>

main()
{
     struct utms mytimes, sample;
     time_t utime, rt, before, after;

     cutimes(&mytimes);                         /* enable execution time
                                                   update feature */

     do {
          rt = rtclock();
          sample = mytimes;                     /* sample mytimes before */
     } while (rt < sample.utms_update);
     before = sample.utms_utime + (rt - sample.utms_update);

     /* Section of code here is where user execution time is to be measured. */

     do {
          rt = rtclock();
          sample = mytimes;                     /* sample mytimes after */
     } while (rt < sample.utms_update);
     after = sample.utms_utime + (rt - sample.utms_update);

     utime = after - before;                    /* compute user time -
                                                   measured in clock ticks */
```

```
      printf("\nCPU time used in user space = %f sec or %ld clock ticks\n",
               (float)utime/(float)CLK_TCK, utime);

      cutimes((struct utms *) 0);                  /* disable execution time
                                                      update feature */
  }
```

SEE ALSO

`exec`(2)

`cpused`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`second`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`dacct` – Enables or disables process and daemon accounting

SYNOPSIS

```
#include <sys/types.h>
#include <sys/accthdr.h>
```

```
int dacct (char *path, int did);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `dacct` system call enables or disables process and daemon accounting.

When process accounting is enabled, an accounting record is written to an accounting file for each process that terminates. Process termination can be caused by an `exit`(2) call, a `chkpnt`(2) call, or receipt of a fatal signal. When a job terminates, an end-of-job record is written.

Similarly, when daemon accounting is enabled, the daemons may write accounting records.

Accounting is disabled when *path* is a null pointer and no errors occur during the system call.

The `dacct` system call accepts the following arguments:

path Points to the path name of the accounting file, which is defined by `acct`(5). The daemon accounting files are defined in `/usr/include/acct/dacct.h`.

did Identifies the type of accounting that is to be enabled or disabled. These daemon identifiers are specified in `/usr/include/sys/accthdr.h`.

If the specified type of accounting is already enabled and *path* differs from the accounting file currently in use, the accounting file will be switched to *path* without the loss of any accounting information.

Only a process with appropriate privilege can use this system call.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the file, the active security label of the process must equal the security label of the file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ACCT`	The process is allowed to use this system call.

`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the file via the permission bits and access control list.
`PRIV_MAC_READ`	The calling process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the file. The super user is allowed to use this system call.

RETURN VALUES

If `dacct` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `dacct` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Search permission is denied on a component of the path prefix.
`EACCES`	The process is not granted write permission to the file.
`EACCES`	The file specified by *path* is not an ordinary file.
`EFAULT`	The *path* argument points to an illegal address.
`EINVAL`	An argument that is not valid was passed to the system call.
`EISDIR`	The *path* argument is a directory.
`EPERM`	The process does not have appropriate privilege to use this system call.
`EROFS`	The specified file resides on a read-only file system.

FILES

`/usr/include/acct/dacct.h`	Defines daemon accounting files
`/usr/include/sys/accthdr.h`	Specifies daemon identifiers

SEE ALSO

`acct`(2), `chkpnt`(2), `exit`(2)

`acct`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`devacct` – Controls device accounting

SYNOPSIS

```
#include <sys/types.h>
#include <sys/acct.h>

int devacct (char *device, int func, int type);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Only a process with appropriate privilege can use this system call. The `devacct` system call accepts the following arguments:

device Specifies the name of a block special device that is a local file system. This name is used only when the `ACCT_LABEL` function is specified.

func This argument can be one of the following:

`ACCT_ON` Turns on accounting for requested type.

`ACCT_OFF` Turns off accounting for requested type.

`ACCT_LABEL` Labels the *device* with the label indicated by *type*. You can label only block special devices.

type Specifies device type. For block special devices, valid values are `0` to `(MAXBDEVNO – 1)`. For character special devices, the values are `0` to `(MAXCDEVNO – 1)` OR'ed with `ACCT_CHSP`. For performance accounting, *type* is `ACCT_PERF` OR'ed with `PERF_01`.

See the `/etc/config/acct_config` file for the block and character device types. `ACCT_PERF` and `PERF_01` are defined in `/usr/include/sys/acct.h`.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

The process must be granted write permission to the device file via the security label. That is, the active security label of the process must equal the security label of the device file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ACCT`	The process is allowed to use this system call.

PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
PRIV_MAC_READ	The calling process is granted search permission to every component of the path prefix via the security label.
PRIV_WRITE	The process is granted write permission to the device file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the file. The super user is allowed to use this system call.

If the PRIV_SU configuration option is enabled, the super user is granted write permission to the device file via the security label.

RETURN VALUES

The devacct system call returns the previous accounting type when called to label a device. It returns the previous state, ACCT_ON or ACCT_OFF, when called to turn device accounting on or off. Otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The devacct system call fails if one of the following error conditions occurs:

Error Code	**Description**
EINVAL	The *type* on the label request is bad.
EINVAL	The device is not a block special on a label request.
EINVAL	The device label is not legal for an on or off request.
EINVAL	The *func* argument is not ACCT_ON, ACCT_OFF, or ACCT_LABEL.
ENODEV	The device is not mounted on a label request.
EPERM	The process does not have appropriate privilege to use this system call.
EPERM	The process is not granted write permission to the file via the security label.

EXAMPLES

The following examples illustrate different uses of the devacct system call. (You also can obtain the functionality in these examples by using the devacct(8) command.)

Example 1: This example shows how to label the block device /dev/dsk/root as a DD–40 device. The numeric type for a DD–40 is 2 because a DD–40 disk drive is associated with the BLOCK_DEVICE2 variable in /etc/config/acct_config.

```
devacct("/dev/dsk/root",ACCT_LABEL,2);
```

Example 2: This example shows how to turn on DD–40 device accounting:

```
devacct(0,ACCT_ON,2);
```

Example 3: This example shows how to turn on performance accounting:

```
devacct(0,ACCT_ON,ACCT_PERF|PERF_01);
```

FILES

/etc/config/acct_config	Accounting configuration file
/usr/include/sys/acct.h	Defines daemon accounting files
/usr/include/sys/param.h	Defines configuration files

SEE ALSO

devacct(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`dmmode` – Sets and gets data migration retrieval mode

SYNOPSIS

```
#include <unistd.h>
int dmmode (int mode);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `dmmode` system call sets the data migration retrieval mode of the calling process. It accepts the following argument:

mode Specifies the mode. `dmmode` set the data migration retrieval mode of the calling process and returns the previous value of the mode. Only the low-order 9 bits of *mode* are used.

A nonzero data migration retrieval mode specifies that offline files are retrieved automatically as soon as they are accessed (see `open`(2)). A value of 0 specifies that files must be explicitly recalled (see `dmget`(1)) before they can be accessed successfully. In this mode, an access attempt on a migrated file results in the return of an error code.

RETURN VALUES

The previous value of the data migration retrieval mode is returned.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `dmmode` system call

SEE ALSO

`open`(2)

`dmget`(1), `dmlim`(1), `dmput`(1) Online only

`dmmctl`(8) Online only

NAME

`dmofrq` – Processes offline file requests

SYNOPSIS

`#include <sys/dmofrq.h>`

`int dmofrq (void *`*fptr*`, int` *cmd*`, void *`*arg*`, struct dmo_hand *`*hand*`,`
`long *`*rcode*`);`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The data migration daemon uses the `dmofrq` system call to process requests related to offline files. Only an appropriately privileged process is allowed to use this system call. You should not use `dmofrq` in your programs; `dmofrq` is intended for use only by data migration packages, such as the Cray Data Migration Facility (DMF).

The `dmofrq` system call accepts the following arguments:

fptr Selects the offline file. The *fptr* argument is either a character pointer to the path name or a word pointer to a `dm_dvino` structure with the device and inode number of the offline file specified.

cmd Selects the type of request to be processed. *cmd* is specified as a character: a lowercase character indicates that *fptr* is a path; an uppercase character indicates that *fptr* is a pointer to a `dm_dvino` structure.

arg Points to an argument required by the command.

hand Specifies the handle for a file to be migrated or recalled.

rcode Specifies a word into which a return code is written by `dmofrq`. The valid return codes and their meanings are defined in `dmofrq.h`.

The *hand* and *rcode* arguments are used only by the migrate and recall commands (`a`, `A`, `b`, `B`, `e`, `E`, `f`, `F`, `g`, `G`, `u`, `U`, `v`, and `V`).

The valid forms of the *cmd* and *arg* arguments are as follows:

`a` or `A` Begins remigration of a dual-state file.

`b` or `B` Begins migration of a file.

`c` or `C` Changes the migration status of a file. The `arg` parameter is a pointer to a `dm_file_change` structure. When the call is processed, if the file matches the `before` and `gen` fields, the migration attributes are changed to the values in the `after` structure. *hand* and *rcode* should be `NULL`.

`d` or `D` Simulates the `secstat`(2) system call. *arg* is a pointer to a `secstat` stucture, and *hand* and *rcode* should be `NULL`.

`e` or `E` Begins remigration of a dual-state file, except that *arg* is a pointer to a `dm_dvino` structure.

`f` or `F` Begins migration of a file, except that the *arg* parameter is a pointer to a `dm_dvino` structure.

`g` or `G` Completes migration of a file. The file is changed to offline and the data blocks are swapped to the destination file and released. *arg* is a pointer to a `dm_dvino` structure.

`l` or `L` Writes the file size (to which *arg* points) to the offline file inode. *arg* is a pointer to a `long`, and *hand* and *rcode* should be `NULL`.

`s` or `S` Simulates the `stat`(2) system call for the indicated file. The status information is returned to the area to which *arg* points. (The lowercase `s` command is obsolete; it will be removed in a future release of UNICOS.) *arg* is a pointer to a `stat` structure, and *hand* and *rcode* should be `NULL`.

`w` or `W` Writes the file handle (to which *arg* points) to the offline file inode. *arg* is a pointer to a `dmo_hand` structure, and *hand* and *rcode* should be `NULL`.

`u` or `U` Completes file recall and zeroes the inode's handle. *arg* is the path of the file containing the disk blocks.

`v` or `V` Completes file recall without zeroing the inode's handle. The handle of the *fptr* file is not cleared; instead, *fptr* becomes a dual-state file with a valid migration handle. *arg* is the path of the file containing the disk blocks.

The `dmofrq` system call receives information from the following structures:

```
typedef struct  dm_dvino {
        dev_t   dm_dev;         /* device number */
        ino_t   dm_ino;         /* inode number */
} dm_dvino_t;

typedef struct  dmo_hand {
        uint    dmport:3;       /* daemon number */
        uint    dmstate:5;      /* file status */
        uint    dmunused1:24;   /* unused */
        uint    machid:32;      /* offline file machine id */
        long    ofilenm;        /* offline file number */
} dmo_hand_t;

typedef struct file_dm_state {
      long    size;             /* File size in bytes */
      long    dm_mid;           /* offline file machine id */
      long    dm_key;           /* offline file number */
      long    dm_state;         /* migration state */
      int     dm_port;          /* daemon number */
} file_dm_state_t;

typedef struct file_dm_change {
      long            gen;    /* File generation number */
```

```
        file_dm_state_t before; /* current state of the file */
        file_dm_state_t after;  /* State of file after the change */
    } file_dm_change_t;
```

The `dm_dvino` structure contains device and inode information, and the `dmo_hand` structure contains handle, state, and port information.

NOTES

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_ADMIN	The process is allowed to use this system call.
PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the file path prefix via the permission bits and access control list.
PRIV_MAC_READ	The process is granted search permission to every component of the file path prefix via the security label.
PRIV_MAC_READ	The process is allowed to override the file security label protections when performing a `stat`(2) or `secstat`(2) operation.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to perform all `dmofrq` operations on any file.

CAUTIONS

You should not use this system call in your programs. It is intended to be used by data migration packages such as DMF only.

RETURN VALUES

If `dmofrq` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error. If `errno` is set to EDMOFRQ, *rcode* will contain a detailed error return code. The valid *rcode* return values and their meanings are defined in the `sys/dmofrq.h` file.

ERRORS

The `dmofrq` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
EBUSY	The file to be changed is in use.
EDMOFF	The data migration system is not configured.
EDMOFRQ	Inconsistent or invalid parameters or file attributes exist. See the value returned to *rcode* for a detailed specification.

`EDMRBPAR`	An invalid *cmd* was specified.
`EDMRNOLF`	The file indicated by *fptr* was not an offline `S_IFOFL` or a regular `S_IFREG` file.
`EDMRNSD`	The *fptr* and *cmd* files were on different file systems.
`EDMRWFT`	The file owning the data blocks was not a regular file.
`EFAULT`	A pointer is not valid.
`EINVAL`	The device and inode do not point to a valid inode.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have appropriate privilege to use this system call.

FILES

`/usr/include/sys/dmofrq.h` Contains definitions related to data migration

SEE ALSO

`dmmode`(2) for information about setting and getting data migration retrieval mode
`secstat`(2) for information about getting file security attributes
`stat`(2) for information about getting file status

NAME

dup – Duplicates an open file descriptor

SYNOPSIS

```
#include <unistd.h>
int dup (int fildes);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The dup system call duplicates an open file descriptor. It accepts the following argument:

fildes Specifies a file descriptor. It is obtained from a creat(2), dup, fcntl(2), open(2), or pipe(2) system call.

The dup system call returns a new file descriptor having the following characteristics in common with the original:

- Same open file (or pipe)
- Same file pointer (that is, both file descriptors share one file pointer)
- Same access mode (read, write, or read/write)

The new file descriptor is set to remain open across exec(2) system calls. See fcntl(2).

The file descriptor returned is the lowest one available.

RETURN VALUES

If dup completes successfully, a nonnegative integer (the file descriptor) is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The dup system call fails if one of the following error conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not a valid open file descriptor.
EMFILE	OPEN_MAX file descriptors are currently open.

FORTRAN EXTENSIONS

The `dup` system call can be called from Fortran as a function:

```
INTEGER fildes, DUP, I
I = DUP (fildes)
```

Alternatively, `dup` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes
CALL DUP (fildes)
```

The Fortran program must not specify both the subroutine call and the function reference to `dup` from the same procedure.

EXAMPLES

The following illustrates how the `dup` system call is used by a shell program to provide for the UNICOS user-level redirection (>) feature.

The user enters the following command to the shell program:

```
$ command > newfile
```

To redirect the output of the command to the named file (`newfile`), the shell closes `stdout` (file descriptor = 1) and then uses `dup` to duplicate the file descriptor for `newfile`. The duplicate file descriptor reuses the file descriptor previously used by `stdout`. The command then executes writing its data to the file having file descriptor 1, which is the file `newfile` rather than the typical `stdout` device.

```
int fd, perms_mask;
char *fptr;

fd = creat(fptr, perms_mask); /* fptr points to requested
                                 file name */
fclose(stdout);
dup(fd);                      /* duplicate file descriptor
                                 replaces stdout */
close(fd);
/* stdout has now been redirected. */
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `dup` system call

SEE ALSO

`close`(2), `creat`(2), `exec`(2), `fcntl`(2), `open`(2), `pipe`(2)

NAME

`exctl` – Exchanges control

SYNOPSIS

```
#include <sys/vm.h>
int exctl (struct vmctxt *pctxt);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `exctl` system call allows a master process to exchange control to a subordinate task running in a virtual machine environment. The subordinate task may be a test kernel, or it may be a user process running under the control of the test kernel. The subordinate task must be fully contained within the address space of the master process. The `exctl` system call accepts the following argument:

pctxt Points to the `vmctxt` structure.

A `vmctxt` structure (virtual machine context) includes the following members:

```
gxp_t     vm_xp;                        /* Exchange package */
int       vm_saveb[MAXBREGS];           /* B registers */
word      vm_savet[MAXTREGS];           /* T registers */
word      vm_savevm;                    /* Vector mask register */
word      vm_savev[MAXVREGS][MAXVLEN];  /* Vector registers */
word      vm_savevm1;                   /* Vector mask register 1 */
word      vm_vsaved;
label_t   vm_save[3];                   /* for switching */
word      ret_status;                   /* Subordinate task's return status */
word      vm_vmsav;                     /* pw_vmsav word (from PWS structure) */
```

The BA and LA values in the exchange package are modified by the system; therefore, these values must be reset each time `exctl` is called. The BA and LA registers are relative to word 0 of the master process. The value of the P register is relative to BA. When this system call is executed, the real kernel converts the given BA and LA values into absolute addresses, loads the hardware registers with the values from the `vmctxt` structure, and starts execution at the given P address. The `exctl` system call returns to the master process when the subordinate task exits for any of the following reasons:

- Normal exchange interrupt
- Error exchange interrupt
- Program range error interrupt
- Operand range error interrupt
- Floating-point error interrupt

- Programmable clock interrupt
- Register parity error

The master process should examine the saved status and the exchange package flags to determine why the subordinate task exited. On return from the `exctl` system call, the `vmctxt` structure contains the contents of the hardware registers as they were when the subordinate task exited.

NOTES

The Programmable Clock Interrupt has been mapped to the signal `SIGALRM`, sent by the `alarm` system call. The intent is to allow the master process to use the `alarm` function to simulate a real-time clock.

RETURN VALUES

If `exctl` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `exctl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	The monitor mode bit (XPM_MM) is set in the exchange package of the `vmctxt` structure of the subordinate process.
`EFAULT`	The context structure to which *pctxt* points is not fully contained in the master process address space.
`EFAULT`	The BA, LA, and P registers in the exchange package are not within the process address space.

SEE ALSO

`alarm`(2)

NAME

`execl`, `execv`, `execle`, `execve`, `execlp`, `execvp` – Executes a file

SYNOPSIS

```
#include <unistd.h>

int execl (const char *path, const char *arg0, const char *arg1,
..., const char *argn, 0);

int execv (const char *path, char *const argv[]);

int execle (const char *path, const char *arg0, const char *arg1,
..., const char *argn, 0, const char *envp[]);

int execve (const char *path, char *const argv[], char *const envp[]);

int execlp (const char *file, const char *arg0, const char *arg1,
..., const char *argn, 0);

int execvp (const char *file, char *const argv[]);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `exec` system call in all its forms transforms the calling process into a new process, which is constructed from an ordinary, executable file called the *new process file*. This file consists of a header and the program images (see `a.out`(5)). There is no return from a successful `exec` because the calling process is overlaid by the new process.

An interpreter file begins with a line of the form:

`#!` *pathname* [*arg*]

The *pathname* argument is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system execs the specified interpreter. The *pathname* specified in the interpreter file is passed on as *arg0* to the interpreter. If *arg* is specified in the interpreter file, it is passed as *arg1* to the interpreter. Any setuid or setgid permissions bits that are set for the interpreter file are ignored. The remaining arguments to the interpreter are *arg0* through *argn* of the file that was originally executed.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

The argument count is *argc,* and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least 1, and the first member of the array points to a string containing the name of the file.

The arguments are as follows:

path Points to a path name that identifies the new process file.

arg0, *arg1*, `...,`
: Points to null-terminated character strings, which constitute the argument list available to the new process. By convention, at least *arg0* must be present, and it must point to a string that is the same as *path* (or its last component).

argv Specifies an array of character pointers to null-terminated strings, which constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). The *argv* argument is terminated by a null pointer.

envp Specifies an array of character pointers to null-terminated strings, which constitute the environment for the new process. The *envp* argument is terminated by a null pointer. For `execl` and `execv`, the C run-time start-up routine (`$start`) places a pointer to the calling process's environment in the global cell, as follows:

```
extern char **environ;
```

It passes the calling process's environment to the new process.

file Points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line:

```
PATH =
```

The environment is supplied by the shell (see `ksh`(1)).

File descriptors open in the calling process remain open in the new process, except for those whose Close-on-exec flag are set; see `fcntl`(2). For file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process are set to terminate the new process. Signals set to be ignored by the calling process are set to be ignored by the new process. Signals set to be caught by the calling process are set to terminate the new process; see `signal`(2).

For signals set by `sigset`(2), `exec` ensures that the new process has the same signal action for each signal type whose action is `SIG_DFL`, `SIG_IGN`, or `SIG_HOLD` as the calling process. However, if the action is to catch the signal, the action will be reset to `SIG_DFL`, and any pending signal for this type will be held.

If the set-user-ID mode bit of the new process file is set (see `chmod`(2)), `exec` sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process will be released and will not be attached to the new process (see `shmat`(2)).

Any additional processes created by `_tfork`(2) or `tfork`(3C) calls are killed off and the resulting program begins execution as a single process. One of the previous multitasked processes may linger on the kernel until the new process exits to maintain the parent-child relationships. This bug condition will be fixed in the next release.

Profiling is disabled for the new process; see `profil`(2). The new process also inherits the following attributes from the calling process:

- Accounting information from `times`(2): `utime`, `stime`, `cutime`, and `cstime`
- Current working directory
- File mode creation mask (see `umask`(2))
- File size limit (see `ulimit`(2))
- Nice value (see `nice`(2))
- Job ID
- Parent process ID
- Process group ID
- Process ID
- Root directory
- `semadj` values (see `semop`(2))
- Time left until an alarm clock signal (see `alarm`(2))
- Trace flag (see `ptrace`(2) request 0)
- tty group ID (see `exit`(2) and `signal`(2))
- Record locks (see `fcntl`(2) and `lockf`(3C))
- Security values: security label, security label range, active and authorized categories.

NOTES

The process must be granted search permission to every component of the path prefix via the permission bits and access control list. The process must be granted search permission to every component of the path prefix via the security label.

The process must be granted execute permission to the file via the permission bits and access control list. The process must be granted execute permission to the file via the security label.

A process with the effective privileges shown are granted the following abilities:

Privilege	Description
PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
PRIV_DAC_OVERRIDE	The process is granted execute permission to the file via the permission bits and access control list.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_READ	The process is granted execute permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted execute permission to the file.

RETURN VALUES

If exec returns to the calling process, an error has occurred; a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The exec system call fails and returns to the calling process if one of the following error conditions occurs:

Error Code	Description
EACCES	The new process file is not a regular file.
EACCES	The new process file mode denies execution permission.
EACCES	Search permission is denied for a directory listed in the new process file's path prefix.
EDMOFF	The file is offline, and the data migration facility is not configured in the system.
EFAULT	The new process file is not as long as indicated by the size values in its header.
EFAULT	The *path*, *argv*, or *envp* argument points to an illegal address.
ENODEV	(CRAY T90 systems only) The new process has requested or requires a CPU type that is not currently available on the system. For example, an exec of a binary built on a CRAY C90 system would result in this error if the CRAY T90 system contained only IEEE CPUs.
ENOENT	One or more components of the new process file's path name do not exist.
ENOEXEC	The exec is not an execlp or execvp, and the new process file has the appropriate access permission but an invalid magic number in its header.

ENOMEM	The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.
ENOTDIR	A component of the new process file's path prefix is not a directory.
EOFFLIN	The file is offline, and automatic file retrieval is disabled.
EOFLNDD	The file is offline, and the data management daemon is not currently executing.
EOFLNNR	The file is offline, and it is currently unretrievable.
E2BIG	The number of bytes in the argument list of the new process is greater than the system-imposed limit of ARG_MAX bytes.

EXAMPLES

The following examples show differences in the use of the various forms of `exec`: `execl`, `execv`, `execle`, `execve`, `execlp`, and `execvp`. In each example, the call overlays the currently executing program with a new program having full path name `/tmp/newprog`. The new program supplies two arguments having values `arg1` and `arg2`.

Example 1: The `execl` system call requires that the new program be specified by a full or relative path name. The argument list passed to the new program is specified as a list of strings in the `execl` request.

The environment existing in the current program is preserved in the new program.

```
execl("/tmp/newprog", "newprog", "arg1", "arg2", 0);
```

Example 2: The `execv` system call requires that the new program be specified by a full or relative path name. The argument list passed to the new program is specified as an array (vector) of pointers to strings in the `execv` request.

The environment existing in the current program is preserved in the new program.

```
static char *arguments[] = {"newprog", "arg1", "arg2", 0};

execv("/tmp/newprog", arguments);
```

Example 3: The `execle` request requires that the new program be specified by a full or relative path name. The argument list passed to the new program is specified as a list of strings in the `execle` request.

The environment existing in the current program is replaced by a new environment in the new program. This environment is specified as an array (vector) of pointers to strings, where each string consists of an environment variable equated to its value.

In this example, the new environment contains only two variables, `ENV1` and `ENV2`. The `execle` request completely replaces the existing environment in the current program with the new environment. Therefore, if the current environment is to preserved with some additional environment variables added, a composite environment must be formed.

```
static char *newenv[] = {"ENV1=string1", "ENV2=string2", 0};

execle("/tmp/newprog", "newprog", "arg1", "arg2", 0, newenv);
```

Example 4: The `execve` system call requires that the new program be specified by a full or relative path name. The argument list passed to the new program is specified as an array (vector) of pointers to strings in the `execve` request.

The environment existing in the current program is replaced by a new environment in the new program. This environment is specified as an array (vector) of pointers to strings, where each string consists of an environment variable equated to its value.

In this example, the new environment contains only two variables, `ENV1` and `ENV2`. The `execve` request completely replaces the existing environment in the current program with the new environment. Therefore, if the current environment is to preserved with some additional environment variables added, a composite environment must be formed.

```
static char *arguments[] = {"newprog", "arg1", "arg2", 0};
static char *newenv[] = {"ENV1=string1", "ENV2=string2", 0};

execve("/tmp/newprog", arguments, newenv);
```

Example 5: The `execlp` request requires that the new program be specified by a file name rather than a full or relative path name (as used in the previous examples). UNICOS looks for the specified file by searching the list of directories included in the user's `PATH` environment variable. As a result of this search, UNICOS could find a program, which is not the intended one, having the requested name. This means the user assumes a greater degree of risk when using the `execlp` request; it is especially dangerous if the calling program is a setuid (set-user-ID) program.

The argument list passed to the new program is specified as a list of strings in the `execlp` request.

The environment existing in the current program is preserved in the new program.

```
execlp("newprog", "newprog", "arg1", "arg2", 0);
```

Example 6: The `execvp` system call requires that the new program be specified by a file name rather than a full or relative path name (as used in previous `exec` examples). UNICOS looks for the specified file by searching the list of directories included in the user's `PATH` environment variable. UNICOS could find a program, which is not the intended one, having the requested name. This means the user must assume a greater degree of risk when using the `execvp` request; it is especially dangerous if the calling program is a setuid (set-user-ID) program.

The argument list passed to the new program is specified as an array (vector) of pointers to strings in the `execvp` request.

The environment existing in the current program is preserved in the new program.

```
static char *arguments[] = {"newprog", "arg1", "arg2", 0};

execvp("newprog", arguments);
```

FILES

/usr/include/unistd.h	Contains C prototype for the exec system call

SEE ALSO

alarm(2), chmod(2), exit(2), fcntl(2), fork(2), nice(2), profil(2), ptrace(2), semop(2), shmat(2), signal(2), sigset(2), _tfork(2), times(2), ulimit(2), umask(2)

ksh(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

lockf(3C), tfork(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

a.out(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`exit`, `_exit`, `newexit`, `_newexit`, `_lwp_exit`, `globalexit`, `_globalexit`, `localexit`, `_localexit`, `_threadexit` – Terminates process

SYNOPSIS

All Cray Research systems:

```
#include <stdlib.h>
void exit (int status);
#include <unistd.h>
void _exit (int status);
```

Cray PVP systems:

```
#include <stdlib.h>
void newexit (int status);
#include <unistd.h>
void _newexit (int status);
void _lwp_exit (int status);
```

Cray MPP systems:

```
#include <stdlib.h>
void globalexit (int status);
void localexit (int status);
#include <unistd.h>
void _globalexit (int status);
void _localexit (int status);
void _threadexit (int status);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `exit`)

DESCRIPTION

The `exit` system call terminates the calling process. It accepts the following argument:

status Specifies the exit status of the process. It is returned to the process's parent process.

The process termination has the following consequences:

- All of the file descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a `wait`(2) system call, it is notified that the calling process has terminated and the low-order 8 bits (that is, bits 0377) of *status* are made available to it; see `wait`(2).
- If the parent process of the calling process is not executing a `wait`(2) system call, the calling process is transformed into a zombie process, which is a process that occupies only a slot in the process table; it has no other space allocated in either the user or the kernel space.
- The parent process ID of all existing child processes and zombie processes of the calling process is set to 1. This means that the initialization process (see `intro`(2)) inherits each of these processes.
- Each attached shared memory segment is detached and the value of `shm_nattch` in the data structure associated with its shared memory identifier is decremented by 1.
- For each semaphore for which the calling process has set a `semadj` value (see `semop`(2)), that `semadj` value is added to the *semval* for the specified semaphore.
- If the process has a process lock, an `unlock` is performed (see `plock`(2)).
- If the process ID, tty group ID, and process group ID of the calling process are equal, the `SIGHUP` signal is sent to each process that has a process group ID equal to that of the calling process.
- If the calling process is the last process in a session to exit, then all nonpersistent IPC facilities created by processes in the session will be removed as if an `IPC_RMID` had been performed on the facility (see `msgget`(2), `semget`(2), and `shmget`(2)).

The C `exit` function may cause cleanup actions before the process exits. The `_exit` function circumvents all cleanup.

On Cray MPP systems, three additional types of exit are available:

`globalexit` or `_globalexit` Forces all PEs into global exit processing, which cleans up all resources in the PEs allocated by the application. The call also causes the Cray PVP system's process to exit.

`globalexit` also executes library clean up routines on the local PE before calling `_globalexit`.

`localexit` or `_localexit` Terminates all threads on the local PE. The last PE to call `_localexit` also initiates a `_globalexit` system call. `_localexit` does not destroy the address space of the application on the calling PE.

`localexit` also executes library clean up routines on the local PE before calling `_localexit`.

`_threadexit` — Terminates the calling thread, either the main user thread or a user protocol thread. The last thread on the PE to call `_threadexit` also initiates a `_localexit` system call.

The `_exit` system call is equivalent to `_localexit`, and `exit` is equivalent to `localexit`. `exit` does library cleanup on the local PE and then calls `_exit`.

NOTES

See the NOTES section on the `signal`(2) man page.

In UNICOS 9.0, a new process model is introduced for multitasked applications. Instead of a multitasked application being considered as multiple processes, an application will be treated as a single process that includes multiple *light-weight processes* (LWPs). Calling `exit` or `_exit` from a multitasking program terminates the entire multitasking group instead of just the calling LWP.

The old behavior for `_exit` will continue to be available through the `_lwp_exit` system call; this is not recommended for general use. `_lwp_exit` is intended only for use by system software.

The `_newexit` and `newexit` system calls were provided to give early access to the new behavior of `_exit` and `exit` in UNICOS 8.0. These will be removed in a subsequent UNICOS release.

FORTRAN EXTENSIONS

The `exit` system call can be called from Fortran as a subroutine:

`CALL EXIT ([`*istat*`])`

The `newexit` system call can be called from Fortran as a subroutine (on all systems except Cray MPP systems):

`CALL NEWEXIT ([`*istat*`])`

The *istat* argument is optional integer exit status. If none is specified, the exit status is 0.

FILES

`/usr/include/sys/stdlib.h`	Contains C prototypes for the `exit`, `newexit`, `globalexit`, and `localexit` system calls
`/usr/include/unistd.h`	Contains C prototypes for the `_exit`, `_newexit`, `_lwp_exit`, `_globalexit`, `_localexit`, and `_threadexit` system calls

SEE ALSO

`intro`(2), `msgget`(2), `plock`(2), `semget`(2), `semop`(2), `shmget`(2), `signal`(2), `wait`(2)

`t_exit`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`fcntl` – Controls open files

SYNOPSIS

```
#include <fcntl.h>
int fcntl (int fildes, int cmd, int arg);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `fcntl` system call provides control over open files as specified by the following arguments:

fildes Specifies an open file descriptor. It is obtained from a `creat`(2), `dup`(2), `fcntl`, `open`(2), or `pipe`(2) system call.

cmd Specifies an action for `fcntl` to perform. The action can be one of the following:

`F_DUPFD` Returns a new file descriptor, as follows:

Lowest-numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (that is, both file descriptors share one file pointer).

Same access mode (read, write, or read/write).

Same file status flags (that is, both file descriptors share the same file status flags).

The Close-on-exec flag associated with the new file descriptor is set to remain open across `exec`(2) system calls.

`F_GETFD` Gets the Close-on-exec flag associated with the *fildes* file descriptor. If the low-order bit is 0, the file remains open across `exec`; otherwise, the file is Closed on execution of `exec`.

`F_SETFD` Sets the Close-on-exec flag associated with *fildes* to the low-order bit of *arg* (0 or 1, as stated previously).

`F_GETFL` Gets file status flags.

`F_SETFL` Sets file status flags to *arg*. Only the `O_APPEND`, `O_NDELAY`, `O_NONBLOCK`, `O_RAW`, and `O_T3D` flags can be set; see `open`(2), `read`(2), and `write`(2).

`F_GETLK` Gets the first lock, which blocks the lock description given by the variable of type `struct flock`, pointed to by *arg*. The retrieved information overwrites the information passed to `fcntl` in the `flock` structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except for the lock type, which will be set to `F_UNLCK`.

`F_SETLK` Sets or clears a file segment lock according to the variable of type `struct flock` pointed to by *arg*. The *cmd* `F_SETLK` establishes read (`F_RDLCK`) and write (`F_WRLCK`) locks and removes either type of lock (`F_UNLCK`). If a read or write lock cannot be set, `fcntl` will return immediately with an error value of –1.

`F_SETLKW` This *cmd* is the same as `F_SETLK`, except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

`F_SETSB` Sets the `sitebits` field in the inode for *fildes* to the value specified in *arg*. The file system type of the file referred to by *fildes* must be `NC1FS`.

`F_SETALF` Adds the allocation flag bits specified by the value of `arg` to any existing allocation flag bits. Valid allocation flag bits are `S_ALF_NOGROW` and `S_ALF_PARTR` (see `stat.h`).

`F_CLRALF` Removes the allocation flag bits specified by the value of *arg* to any existing allocation flag bits. Valid allocation flag bits are `S_ALF_NOGROW` and `S_ALF_PARTR` (see the `stat.h` file).

`F_GETXT` Gets address extent information from the inode for *fildes*. The calling process must be the owner of the file or have appropriate privilege.

`F_RSETLK`
`F_RSETLKW`
`F_RGETLK` Used by the network lock daemon, `lockd`(8), to communicate with the NFS server kernel to handle locks on NFS files. Specify these options at your own risk. A file lock may be removed if you use them.

arg Specifies a value that varies in meaning according to the action specified by *cmd*.

This number indicates the file descriptor if *cmd* is `F_DUPFD`, the Close-on-exec flag if *cmd* is `F_SETFD`, the File Status flag if *cmd* is `F_SETFL`, the address of a variable of type `struct flock` if *cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW`, the one word octal `sitebits` value if *cmd* is `F_SETSB`, or address of a variable of type `struct xt_report` if *cmd* is `F_GETXT`.

A read lock prevents any process from placing a write lock on the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from placing a read lock or write lock on the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The `flock` structure describes the type (`l_type`), starting offset (`l_whence`), relative offset (`l_start`), size (`l_len`), process ID (`l_pid`), and system ID (`l_sysid`) of the segment of the file to be affected. The process ID and system ID fields are used only with the `F_GETLK` *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting `l_len` to 0. If such a lock also has `l_whence` and `l_start` set to 0, the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a `fork`(2) system call.

When mandatory file and record locking are active on a file (see `chmod`(2)), `read`, and `write` system calls issued on the file are affected by the record locks in effect.

If *cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW` and the file is located on a UNICOS shared file system (SFS), *cmd* affects the entire file, and cannot be used to specify areas of the file. If locks are cleared with either `F_SETLK` or `F_SETLKW` and `F_UNLCK`, an `O_EXCL` open lock is also cleared.

If *cmd* is `F_SETSB`, *arg* contains the single word `sitebits` value.

If *cmd* is `F_GETXT`, *arg* contains the address of the variable of type `struct xt_report`. The calling process supplies the number of extents to return in the `xtr_size` field of the *arg* variable. Information returned by `fcntl` includes the total number of extents (`xtr_nextent`) for the file, the number of indirect blocks (`xtr_nindirs`), the number of data blocks (`xtr_nblocks`), the number of data blocks in primary partitions (`xtr_pblocks`), the number of data blocks in secondary partitions (`xtr_sblocks`) of the file system, the minimum (`xtr_minblks`) and the maximum (`xtr_maxblks`) data blocks in a single extent, and the data extents (`xtr_xtnt`) for the file. The data extent fields contain the starting block and number of contiguous blocks, each as a 32-bit field.

NOTES

In the future, the `errno` variable will be set to `EAGAIN` rather than `EACCES` when a section of a file is already locked by another process; therefore, portable application programs should expect and test for either value.

If `F_GETLK` is specified, the process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

If `F_SETLK`, `F_SETLKW`, `F_SETSB`, `F_SETALF`, or `F_CLRALF` is specified, the process must be granted write permission to the file via the security label. That is, the active security label of the process must equal the security label of the file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_FOWNER`	The process is considered the owner of the file.
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

`PRIV_MAC_WRITE` The process is granted write permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted read and write permission to the file via the security label and is considered the owner of the file.

RETURN VALUES

If `fcntl` completes successfully, the value returned depends on the *cmd* argument. The value returned is as follows:

`F_DUPFD`	A new file descriptor
`F_GETFD`	Value of flag (only the low-order bit is defined)
`F_SETFD`	Value other than –1
`F_GETFL`	Value of file flags
`F_SETFL`	Value other than –1
`F_GETLK`	Value other than –1
`F_SETLK`	Value other than –1
`F_SETLKW`	Value other than –1
`F_SETSB`	Value other than –1
`F_SETALF`	Value of the allocation flags for the file prior to request
`F_CLRALF`	Value of the allocation flags for the file prior to request
`F_GETXT`	Value other than –1

Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `fcntl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	The *cmd* argument is `F_SETLK`, the type of lock (`l_type`) is a read (`F_RDLCK`) lock, and the segment of a file to be locked is already write locked by another process, or the type is a write (`F_WRLCK`) lock, and the segment of a file to be locked is already read or write locked by another process.
`EBADF`	The *fildes* argument is not a valid open file descriptor.
`EBADF`	The requested command is `F_GETLK` and the calling process does not have MAC read access to the file which the file descriptor refers, or the command was `F_SETLK`, `F_SETLKW`, `F_SETALF`, or `F_CLRALF`, and the calling process does not have MAC write access to the file to which the file descriptor refers.

EDEADLK	The *cmd* argument is F_SETLKW, the lock is blocked by a lock from another process, and it would cause a deadlock to put the calling process to sleep and wait for that lock to become free.
EFAULT	The *cmd* argument is F_SETLK, or F_GETXT and *arg* points outside the program address space.
EINTR	A signal was caught during the fcntl system call.
EINVAL	The *cmd* argument is F_DUPFD, and *arg* is negative, greater than, or equal to the maximum number of files per process (OPEN_MAX).
EINVAL	The *cmd* argument is F_GETLK, F_SETLK, or SETLKW, and *arg* or the data to which it points is not valid.
EINVAL	The *fildes* argument refers to a file on a foreign file system (for example, NFS).
EINVAL	The *cmd* is F_SETSB, and the *fildes* argument does not refer to a regular file (for example, a directory).
EINVFS	The *fildes* argument refers to a file on a file system that is not a NC1FS file system.
EMFILE	The *cmd* argument is F_DUPFD, and NOFILE file descriptors are currently open.
ENOLCK	The *cmd* argument is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there is no space for additional record locks to be set (too many file segments locked) because the system maximum has been exceeded.
EPERM	The *cmd* argument is F_GETXT, and the calling process is not the owner of the file and does not have appropriate privilege.

FORTRAN EXTENSIONS

The fcntl system call can be called from Fortran as a function:

```
INTEGER fildes, cmd, arg, FCNTL, I
I = FCNTL (fildes, cmd, arg)
```

Alternatively, fcntl can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes, cmd, arg
CALL FCNTL (fildes, cmd, arg)
```

The Fortran program must not specify both the subroutine call and the function reference to fcntl from the same procedure.

EXAMPLES

The following examples illustrate different uses of the `fcntl` system call.

Example 1: The file `datafile` is opened for writing only, and the current offset into the file is set to the beginning of the file since the `O_APPEND` flag was not specified on the `open` request.

Later in the program, it is desirable for all writes to append onto the end of the file. After first obtaining the current file status flags using the `F_GETFL` command, the `fcntl` system call sets the `O_APPEND` option with the `F_SETFL` command. All writes to `datafile` extend the file.

```
int fd, flags;

fd = open("datafile", O_WRONLY);

flags = fcntl(fd, F_GETFL, 0);       /* get current file status flags */
fcntl(fd, F_SETFL, flags | O_APPEND);/* add append mode for the file */
```

Example 2: By default, any open file remains open in the new program when an `exec` system call overlays the current program with a new program. (Some system calls in the example are not supported on Cray MPP systems.)

The `myfile` file is open when the `execl` request is made. However, the `fcntl` system call with the `F_SETFD` command sets the `close-on-exec` flag to value `1` before the `execl` request is issued. Therefore, `myfile` is closed when the `execl` request is made, and it does not remain open in the new program named `newprog`.

```
int fd;

fd = open("myfile", O_RDONLY);

fcntl(fd, F_SETFD, 1);     /* file myfile should close on exec(2) */

execl("/tmp/newprog", "newprog", "arg1", "arg2", 0);
```

Example 3: This example shows how `fcntl` safely updates a specific record of a file. The first `fcntl` request sets a write lock on the tenth record of `dbfile`. This record is then read, modified in memory, and rewritten back to the device. The second `fcntl` request unlocks the locked record.

Because `fcntl` uses `F_SETLK`, the request fails immediately if any other process has a lock set that blocks setting this lock.

```
struct flock lock;
int fd, recsize;

fd = open("dbfile", O_RDWR);

lseek(fd, 9 * recsize, 0);/* seek to 10th record in the file */

lock.l_type = F_WRLCK;    /* set write lock */
lock.l_whence = 1;        /* starting offset = current location */
lock.l_start = 0;         /* relative offset = current location */
lock.l_len = recsize;     /* # of bytes to lock - one record */

if (fcntl(fd, F_SETLK, &lock) == -1) {
     perror("record locking operation failed");
     exit(1);
}

/* Code here is to update file record including read and write operations. */

lseek(fd, 9 * recsize, 0);/* seek again to 10th record in the file */

lock.l_type = F_UNLCK;    /* set to unlock */
lock.l_whence = 1;        /* starting offset = current location */
lock.l_start = 0;         /* relative offset = current location */
lock.l_len = recsize;     /* # of bytes to unlock - one record */

if (fcntl(fd, F_SETLK, &lock) == -1) {
     perror("record unlocking operation failed");
     exit(1);
}
```

Example 4: The following `fcntl` system call opens and write locks the entire file (named `datafile`). The `F_SETLKW` command to the `fcntl` request causes the program to sleep if there is any other lock set on the file that causes this lock to be blocked.

When a blocking lock is released, this lock proceeds.

```
struct flock lock;
int fd;

fd = open("datafile", O_RDWR);

lock.l_type = F_WRLCK;       /* set write lock */
lock.l_whence = 1;           /* starting offset = current location */
lock.l_start = 0;            /* relative offset = current location */
lock.l_len = 0;              /* # of bytes to lock - to EOF */

if (fcntl(fd, F_SETLKW, &lock) == -1) {
     perror("record locking operation failed");
     exit(1);
}
```

Example 5: The `fcntl` request with `F_SETLK` command attempts to lock the entire file (named `lockfile`). If this attempt fails, an additional `fcntl` request with command `F_GETLK` determines the process ID of the process that currently holds a lock on the file.

```
int fd;
struct flock lock;

fd = open("lockfile", O_RDONLY);

lock.l_type = F_RDLCK;        /* set read lock */
lock.l_whence = 1;            /* starting offset = current location */
lock.l_start = 0;             /* relative offset = current location */
lock.l_len = 0;               /* # of bytes to lock -> to EOF */
lock.l_pid = 0;               /* initialize pid for later */

if (fcntl(fd, F_SETLK, &lock) == -1) {
     perror("file locking operation failed");
     fcntl(fd, F_GETLK, &lock);/* who’s got it locked ? */
     if (lock.l_pid != 0) {
          printf("process having file ’lockfile’ locked = %d\n",
               lock.l_pid);
     }
     lock.l_type = F_RDLCK;   /* set read lock again since */
                              /*  F_GETLK resets to F_UNLCK */
}
```

FILES

`/usr/include/fcntl.h`	Contains symbol definitions for the `fcntl` system call
`/usr/include/sys/stat.h`	Contains definitions of `S_ALF_NOGROW` and `S_ALF_PARTR`

SEE ALSO

`chmod`(2), `close`(2), `creat`(2), `dup`(2), `exec`(2), `fork`(2), `open`(2), `pipe`(2), `read`(2), `write`(2)

`fcntl`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`lockd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

NAME

`fgetpal` – Gets the privilege assignment list (PAL) and privilege sets of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int fgetpal (int fdes, pal_t *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `fgetpal` system call gets the privilege assignment list (PAL) and privilege sets of the file identified by a file descriptor and returns the information in the buffer.

The `fgetpal` system call accepts the following arguments:

fdes Specifies file descriptor that identifies the file for which the PAL and privilege sets are retrieved.

buf Contains pointer to the return buffer.

bufsize Indicates the maximum size of the buffer in bytes.

The calling process must have MAC read access to the file or have `PRIV_MAC_READ` in its effective privilege set.

RETURN VALUES

If `fgetpal` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `fgetpal` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	The caller is denied MAC read access to the file.
`EBADF`	The supplied file descriptor is invalid.
`EFAULT`	The *buf* argument points outside the address space of the process.
`EINVAL`	The *bufsize* argument specifies an invalid value.

SEE ALSO

`fsetpal`(2), `getpal`(2), `setpal`(2)

NAME

`fork` – Creates a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `fork` system call creates a new process (child process), which is an exact copy of the calling process (parent process). This means that the child process inherits the following attributes from the parent process:

- Environment
- Close-on-exec flag (see `exec`(2))
- Signal handling settings
- Set-user-ID mode bit
- Set-group-ID mode bit
- Profiling on/off status
- *nice* value (see `nice`(2))
- Process group ID
- Job ID
- tty group ID (see `exit`(2) and `signal`(2))
- Trace flag (see `ptrace`(2) *request* `0`)
- Current working directory
- Root directory
- File mode creation mask (see `umask`(2))
- File size limit (see `ulimit`(2))
- All attached shared memory segments (see `shmat`(2))

The child process differs from the parent process in the following ways:

- The child process has a unique process ID.
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.
- Process locks are not inherited by the child process (see `plock`(2)).
- The `utime`, `stime`, `cutime`, and `cstime` of the child process are set to 0. The time left until an alarm clock signal is reset to 0.
- Record locks set by the parent process are not inherited by the child process (see `fcntl`(2) and `lockf`(3C)).
- In a multitasking group, only the process that executed the `fork` system call is copied.
- Each attached shared memory segment is attached and the value of `shm_nattch` in the data structure associated with the shared memory segment is incremented by 1.
- All `semadj` values are cleared (see `semop`(2))

NOTES

The child process inherits all active and authorized security attributes from the parent process. These attributes include levels, compartments, categories, privileges, and privilege text.

RETURN VALUES

If `fork` completes successfully, it returns a value of 0 to the child process and returns the process ID of the child process to the parent process; otherwise, a value of –1 is returned to the parent process, no child process is created, and `errno` is set to indicate the error.

ERRORS

The `fork` system call fails and no child process is created if one of the following error conditions occurs:

Error Code	**Description**
`EAGAIN`	The system-imposed limit on the total number of processes under execution in the whole system (`NPROC`) is exceeded.
`EAGAIN`	The system-imposed limit on the total number of processes under execution by one user (`CHILD_MAX`) is exceeded.
`EBUSY`	This error also occurs if you try to enable accounting when it is already enabled or if you issue a `restart`(2) attempt when another job or process in the system is using the *jid* or any *pid* associated with the job (or process) to be restarted.

EINTR — An asynchronous signal (such as `interrupt` or `quit`), which you have elected to catch, occurred during a `fork` system call. When execution resumed after processing the signal, the interrupted system call returned this error condition.

EMEMLIM — More memory space was requested than is allowed for the processes attached to this lnode. The maximum value is set by the `-c` option of the `shradmin`(8) command. This error appears only on systems running the fair-share scheduler.

ENOEXEC — A request was made to execute a file that, although it has the appropriate permissions, does not start with a valid magic number (see `a.out`(5)).

ENOMEM — During an `exec`(2) or `sbreak`(2) system call, a program requested more space than the system could supply. This is not a temporary condition; the maximum space specification is a system parameter.

EPROCLIM — More processes were requested than is allowed for this lnode. The maximum value is set by the `-p` option of the `shradmin`(8) command. This error appears only on systems running the fair-share scheduler.

FORTRAN EXTENSIONS

The `fork` system call can be called from Fortran as a function:

```
INTEGER FORK, I
I = FORK ()
```

EXAMPLES

The following examples illustrate different uses of the `fork` system call.

Example 1: The `fork` request generates a new process that executes the same program as the parent process. The program executing in the parent process that issued the `fork` request is also executing in the child process at the completion of the request. (Usually, in application programs, when a parent generates a child process, the programmer intends for the child process to execute a different program than the one executing in the parent process as illustrated in example 2.)

The return value from `fork` indicates whether execution is in the parent or child process.

```
pid_t res;

if ((res = fork()) == -1) {
     perror("fork failed");
     exit(1);
}

if (res == 0) {
     /* code here is executed in the child process */
}
else {
     /* code here is executed in the parent process */
}
```

Example 2: In many cases, when a parent generates a child process, the programmer wants a different program to execute in the child process rather than the same program as in the parent process. Therefore, when the child process returns from `fork`, it immediately issues an `exec`(2) system call.

In this example, the newly created child process performs an `execl`(2) request to load a different program (`childprog`) for execution. The parent and child processes then execute different programs in parallel.

```
pid_t res;

if ((res = fork()) == -1) {
     perror("fork failed");
     exit(1);
}

if (res == 0) {            /* In child process? */
     execl("childprog", "childprog", "arg1", "arg2", 0);
     perror("exec for childprog failed");
     _exit(1);
}

/* parent process continues execution here */
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `fork` system call

SEE ALSO

brk(2), exec(2), execl(2), exit(2), fcntl(2), nice(2), plock(2), ptrace(2), restart(2), sbreak(2), semop(2), shmat(2), signal(2), times(2), ulimit(2), umask(2), wait(2)

lockf(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR−2080

a.out(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR−2014

shradmin(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR−2022

NAME

`fsetpal` – Sets the privilege assignment list (PAL) and privilege sets of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int fsetpal (int fdes, pal_t *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `fsetpal` system call sets the privilege assignment list (PAL) and privilege sets of the file identified by a file descriptor using the information in the buffer.

The `fsetpal` system call accepts the following arguments:

fdes Specifies file descriptor that identifies the file for which the PAL and privilege sets are set.

buf Contains pointer that contains information.

bufsize Indicates the maximum size of the buffer in bytes.

The calling process must have `PRIV_SETFPRIV` in its effective privilege set, and must either be the file's owner or have `PRIV_FOWNER` in its effective privilege set. The calling process must have MAC write access to the file or have `PRIV_MAC_WRITE` in effective privilege set. The calling process can change the state of privileges in the file's allowed, forced, or set-effective privilege sets only when those privileges are also in the caller's permitted privilege set.

If the `PRIV_SU` option is enabled, any process with effective user ID 0 meets all the requirements specified in the previous paragraph.

RETURN VALUES

If `fsetpal` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `fsetpal` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EBADF`	The supplied file descriptor is invalid.
`EBADF`	The process is not granted MAC write permission to the file and does not have appropriate privilege.

`EFAULT`	The *buf* argument points outside the address space of the process.
`EINVAL`	The *bufsize* argument specifies an invalid value.
`EINVAL`	The contents of the supplied PAL is invalid.
`EPERM`	The process is not the file owner, and does not have appropriate privilege.
`EROFS`	The affected file system is a read-only file system.
`ESECADM`	The process does not have appropriate privilege to use this system call.

SEE ALSO

`fgetpal`(2), `getpal`(2), `setpal`(2)

NAME

`fsync` – Synchronizes the in-core state of a file with that on disk

SYNOPSIS

```
#include <unistd.h>
int fsync (int fildes);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `fsync` system call moves all modified data and attributes of a file descriptor to a permanent storage device. It accepts the following argument:

fildes Specifies the file descriptor.

All in-core modified copies of buffers for the associated file have been written to a disk when the call returns. Programs requiring that a file be in a known state should use this call.

In contrast, the `sync`(2) system call schedules disk I/O for all files (as if an `fsync` system call had been done on all files), but returns before the I/O completes.

RETURN VALUES

When `fsync` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `fsync` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EBADF`	The value of *fildes* is not a valid file descriptor.
`EINVAL`	File descriptor *fildes* refers to a socket, not a file.
`EIO`	An I/O error occurred during a read from or a write to the file system.

EXAMPLES

The following example shows how to use the `fsync` system call to ensure that a device is updated before a file is closed and a process exits. Because the `O_RAW` flag is not specified in the `open`(2) request, file `datafile` is opened by use of the buffered I/O method. When write operations update `datafile`, these changes are made in system cache buffers and the actual updates to the device are delayed.

Completion of the `fsync` request assures the user that these changes have reached the device(s) before `datafile` is closed and the process exits.

```
#include <fcntl.h>

main()
}
     int fd;

     fd = open("datafile", O_RDWR);

     /* updates to file "datafile" performed here */

     if (fsync(fd) == -1) {  /* insure that data arrives on device */
          perror("fsync failed");
          exit(1);
     }

     close(fd);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `fsync` system call

SEE ALSO

`open`(2), `sync`(2)

`cron`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

getash – Gets an array session handle

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

ash_t getash (void);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The getash system call returns the array session handle for the array session that contains the calling process.

The handle for an array session is assigned when the array session is first created. This handle can be overridden using the privileged setash(2) system call.

RETURN VALUES

The getash system call always returns the array session handle. There are no error conditions.

SEE ALSO

newarraysess(2), setash(2)

array_services(7), array_sessions(7)

NAME

`getdents` – Reads and formats directory entries as file system independent

SYNOPSIS

```
#include <sys/dirent.h>
int getdents (int fildes, char *buf, unsigned nbyte);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getdents` system call tries to read a specified number of bytes from a directory and to format them as file system-independent directory entries in a buffer. Because the file system-independent directory entries vary in length, usually the actual number of bytes returned is strictly less than the number of bytes specified.

The `getdents` system call accepts the following arguments:

fildes Specifies a file descriptor associated with a directory. It is obtained from an `open`(2) or `dup`(2) system call.

buf Points to the buffer.

nbyte Specifies the number of bytes.

The file system-independent directory entry is specified by the `dirent` structure (see `dirent`(5)).

On devices capable of seeking, `getdents` starts at a position in the file given by the file pointer associated with *fildes.* On return from `getdents`, the file pointer is incremented to point to the next directory entry.

This system call was developed to implement the `readdir`(3C) routine (for a description see `directory`(3C)), and it should not be used for other purposes.

NOTES

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If `getdents` completes successfully, a nonnegative integer is returned, indicating the number of bytes actually read. A value of 0 indicates that the end of the directory has been reached. If the system call fails, a −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getdents` system call fails if one of the following conditions occurs:

Error Code	Description
`EBADF`	The *fildes* argument is not a valid file descriptor open for reading.
`EBADF`	The active security label of the process is not greater than or equal to the security label of the directory, and the process does not have appropriate privilege.
`EFAULT`	The *buf* argument points outside the allocated address space.
`EINVAL`	The *nbyte* argument is not large enough for one directory entry.
`EIO`	An I/O error occurred while the file system was being accessed.
`ENOENT`	The current file pointer for the directory is not located at a valid entry.
`ENOTDIR`	The *fildes* argument is not a directory.

SEE ALSO

`dup`(2), `open`(2)

`directory`(3C), `readdir`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`dirent`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`getdevn` – Gets device number or driver entry

SYNOPSIS

`int getdevn (unsigned long` *number*`, int` *bflag*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getdevn` system call gets a device number or driver entry. It accepts the following arguments:

number Indicates major device number or driver name. If *number* is a major device number, `getdevn` returns the driver name as a character constant; If *number* is larger than the largest major number, `getdevn` assumes that it is a character constant and returns the major number for that device.

bflag Indicates device type. If *bflag* is specified (nonzero), the device is assumed to be a block device; if it is 0, the device is a character device.

RETURN VALUES

When `getdevn` completes successfully, a nonnegative value is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getdevn` system call fails if the following error condition occurs:

Error Code	**Description**
`EINVAL`	The arguments specify a device number that is undefined or a driver entry that is undefined.

EXAMPLES

The following examples show how to use the `getdevn` system call.

Example 1: In this example, `getdevn` returns the major number of the block disk driver:

```
i = getdevn ('dev_dd',1);
```

Example 2: In this example, `getdevn` returns the major number of the `hi.c` driver:

```
i = getdevn ('dev_hppi',0);
```

NAME

`getfacl` – Gets access control list entries for file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/acl.h>

int getfacl (char *fname, struct acl *aclents, int count);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getfacl` system call gets the access control list (ACL) entries of a file and stores them in an array.

The `getfacl` system call accepts the following arguments:

fname Specifies the file that contains ACL entries.

aclents Specifies the array.

count Indicates the maximum number of ACL entries that can be put in the array.

NOTES

Errors are recorded in the security log if discretionary access control logging is enabled.

The maximum number of ACL entries supported is defined by `ACLSIZE` in `acl.h`.

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and ACL.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label.
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted read permission to the file via the security label.

RETURN VALUES

If `getfacl` completes successfully, the number of entries in the file's ACL is returned (depending on the value of *count*, this may not necessarily be the number of entries returned in the array *aclents*); otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getfacl` system call fails if one of following error conditions occurs:

Error Code	**Description**
`EACCES`	Process does not have mandatory read access to the file.
`EFAULT`	The *fname* argument points outside the process address space.
`EFAULT`	The *aclents* argument resides outside the process' address space.
`EINVAL`	The *count* argument is less than 0.
`EMANDV`	The process is denied read permission to the file via the security label.
`ENAMETOOLONG`	The supplied file name is too long.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.

EXAMPLES

This example shows how to use the `getfacl` system call to retrieve ACL entries for a file. That is, the `getfacl` request retrieves the ACL entries for file `datafile`. Then, these entries are displayed on `stdout`.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/acl.h>
#include <pwd.h>

main(int argc, char *argv[])
{
     struct acl buf[ACLSIZE];
     struct passwd *pwptr;
     int no, i;

     if ((no = getfacl("datafile", buf, ACLSIZE)) == -1) {
          perror("getfacl failed");
          exit(1);
     }

     printf("Access control list for datafile contains ");
     printf("the following users:\n\n");
     printf("ID         Logon ID   Name");
     printf("                        Permissions\n\n");

     for (i = 0; i < no; i++) {
          pwptr = getpwuid(buf[i].ac_usid);
          printf("%-5d      %-10s %-25s   %c%c%c\n",
                  buf[i].ac_usid, pwptr->pw_name, pwptr->pw_gecos,
                  buf[i].ac_mode & 04 ? 'r' : ' ',
                  buf[i].ac_mode & 02 ? 'w' : ' ',
                  buf[i].ac_mode & 01 ? 'x' : ' ');
     }
}
```

FILES

/usr/include/sys/acl.h	Defines access control list structure

SEE ALSO

`rmfacl`(2), `secstat`(2), `setfacl`(2)

`spacl`(1), `spclr`(1), `spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `acl`(5), `dirent`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

getgroups, setgroups – Gets or sets group list

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <unistd.h>

int getgroups (int ngroups, gid_t *gidset);

int setgroups (int ngroups, gid_t *gidset);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to getgroups)

DESCRIPTION

The getgroups system call gets the current group list of the user process and stores it in the *gidset* array. It accepts the following arguments:

ngroups Indicates the maximum number of entries that may be placed in *gidset*. No more than NGROUPS_MAX, as defined in the sys/param.h file, are ever returned.

gidset Points to the array.

The setgroups system call sets the group list of the current user process according to the *gidset* array. The *ngroups* argument indicates the number of entries in the array, and it must be no more than NGROUPS_MAX, as defined in the sys/param.h file. Only a process with appropriate privilege can set its group list.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_SETGID	The process is allowed to set its group list.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_ID permbit is allowed to set its group list.

RETURN VALUES

If `getgroups` completes successfully, `getgroups` returns the number of groups to which the user process belongs; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

If `setgroups` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getgroups` or `setgroups` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EFAULT`	The *gidset* argument specifies an address that was not valid.
`EINVAL`	The *ngroups* argument is smaller than the number of groups to which the user process belongs.
`EPERM`	The process does not have appropriate privilege to set its group list.

EXAMPLES

This example shows how to use the `getgroups` system call to retrieve the list of valid groups for the current process. That is, the `getgroups` request retrieves the list for the current process. Then, the list is displayed on `stdout`.

`NGROUPS_MAX`, defined in the `sys/param.h` file, specifies the maximum number of groups in which a user can be a member.

```
gid_t group[NGROUPS_MAX];
int no, i;

if ((no = getgroups(NGROUPS_MAX, group)) == -1) {
     perror("getgroups failed");
     exit(1);
}

printf("The current user belongs to the following groups:\n\n");
for (i = 0; i < no; i++) {
     printf("%d\n", group[i]);
}
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/sys/param.h`	Defines configuration files

`/usr/include/unistd.h`	Contains C prototype for the `getgroups` and `setgroups` system calls

SEE ALSO

`initgroups`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`gethostid`, `sethostid` – Gets or sets unique identifier of local host

SYNOPSIS

```
#include <unistd.h>
int gethostid (void);
int sethostid (int hostid);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `gethostid` system call returns the 32-bit identifier for the local host.

The `sethostid` system call establishes a 32-bit identifier, which is intended to be unique among identifiers of all existing systems running UNIX or UNICOS software. This identifier is usually the DARPA Internet address for the local host. Only a process with appropriate privilege can use this call; it is typically performed at boot time.

The `sethostid` system call accepts the following argument:

hostid Specifies the host identifier.

Before `sethostid` is called, the default identifier value is 0.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to set the host ID.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to set the host ID.

RETURN VALUES

The `gethostid` system call returns the host ID.

If `sethostid` completes successfully, 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `sethostid` system call fails if the following error condition occurs:

Error Code	Description
`EPERM`	The process does not have appropriate privilege to set the host ID.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `gethostid` and `sethostid` system calls

SEE ALSO

`gethostname`(2)

`hostid`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`gethostname`, `sethostname` – Gets or sets name of local host

SYNOPSIS

```
#include <unistd.h>
int gethostname (char *name, int namelen);
int sethostname (char *name, int namelen);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `gethostname` system call returns the official host name for the local host, as previously set by `sethostname`. The returned name is null-terminated, unless insufficient space is provided.

The `sethostname` system call sets the name of the host machine. This call is restricted to a process with appropriate privilege and is normally used only when the network is initialized. Before `sethostname` is called, the default host name is the null string.

The `gethostname` and `sethostname` system calls accept the following arguments:

name Points to the address of an array of bytes where the name is to be stored (`gethostname`) or is stored (`sethostname`).

namelen Specifies the size of the *name* array. This length is measured in bytes.

NOTES

If *namelen* is less than the length of the host's official name, the official name is truncated to fit.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to set the host name.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to set the host name.

RETURN VALUES

The `gethostname` system call returns the host name.

If `sethostname` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `gethostname` or `sethostname` system call fails if one of the following conditions occurs:

Error Code	Description
EFAULT	For the `gethostname` or `sethostname` system call, the *name* or *namelen* argument gave an address that was not valid.
EPERM	The process does not have appropriate privilege to set the host name.

BUGS

For the `sethostname` system call, host names are limited to `MAXHOSTNAMELEN` (defined in the `/usr/include/sys/param.h` file).

EXAMPLES

The following example shows how to use the `gethostname` system call to retrieve the official host name for the local host. This `gethostname` request finds the official name of the local host and places it in the array `hostname` as a null-terminated character string.

```
#define HOSTNM_MAX  256

char hostname[HOSTNM_MAX];

if (gethostname(hostname, HOSTNM_MAX) == -1) {
     perror("gethostname failed");
     exit(1);
}
else {
     /* official host name for local host now
        resides in array hostname */
}
```

FILES

/usr/include/sys/param.h	Defines configuration files
/usr/include/unistd.h	Contains C prototype for the `gethostname` and `sethostname` system calls

SEE ALSO

`gethostid`(2)

`hostname`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`getinfo` – Gets specified user, job, or process signal information

SYNOPSIS

```
#include <sys/getinfo.h>
int getinfo (int type, int id, char *arg, int size);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getinfo` system call obtains detailed information from the kernel about why a `SIGINFO` signal occurred for a process.

type Specifies the type of request. Currently, `GI_SIGINFO` is the only valid *type*.

If *type* = `GI_SIGINFO`, `getinfo` returns a mask. The *id* argument is ignored, the *arg* argument is the address of a `long` integer that will receive the `SIGINFO` mask, and *size* is `sizeof(long)`. The `SIGINFO` mask is defined in `sys/getinfo.h`, and it consists of a bit for every possible reason a `SIGINFO` signal could occur. The bits in the mask are, as follows:

`RESERVED_0`	(1<< 0)	Reserved
`RESERVED_1`	(1<< 1)	Reserved
`SIGINFO_AFQL`	(1<< 2)	Account file quota limit
`SIGINFO_AFQW`	(1<< 3)	Account file quota warning
`SIGINFO_AIQL`	(1<< 4)	Account inode quota limit
`SIGINFO_AIQW`	(1<< 5)	Account inode quota warning
`RESERVED_6`	(1<< 6)	Reserved
`RESERVED_7`	(1<< 7)	Reserved
`SIGINFO_GFQL`	(1<< 8)	Group file quota limit
`SIGINFO_GFQW`	(1<< 9)	Group file quota warning
`SIGINFO_GIQL`	(1<<10)	Group inode quota limit
`SIGINFO_GIQW`	(1<<11)	Group inode quota warning
`RESERVED_12`	(1<< 12)	Reserved
`RESERVED_13`	(1<< 13)	Reserved
`SIGINFO_UFQL`	(1<<14)	User file quota limit
`SIGINFO_UFQW`	(1<<15)	User file quota warning
`SIGINFO_UIQL`	(1<<16)	User inode quota limit
`SIGINFO_UIQW`	(1<<17)	User inode quota warning
`SIGINFO_MIG`	(1<<27)	A file is being migrated online
`SIGINFO_PSLW`	(1<<40)	Process soft limit warning

SIGINFO_SSLW	(1<<41)	Session soft limit warning
SIGINFO_USLW	(1<<42)	User soft limit warning

id Specifies 0. 0 is the only valid value. If specified, the *id* argument is ignored; the request is always performed for the calling process.

arg Points to an argument dependent on *type*.

size Specifies the size of *arg* in characters. This is intended to provide interface compatibility if future enhancements are needed.

NOTES

The active security label of the process must be greater than or equal to the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is allowed to override security label restrictions.
PRIV_POWNER	The process is considered the owner of every affected process.

If the PRIV_SU configuration option is enabled, the super user is considered the owner of every affected process. If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions.

RETURN VALUES

If getinfo completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The getinfo system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The *size* argument is smaller than the size needed to return the data.
EFAULT	The *arg* argument points outside the allocated address space.
EINVAL	The *type* argument is an invalid request.

EXAMPLES

UNICOS supports inode and file size quota enforcement. This example shows how to use the `getinfo` system call to determine what type of quota is reached during the execution of a process.

```
#include <signal.h>

main()
{
     void handler(int signo);

     signal(SIGINFO, handler);

     /* If, during execution of this process, an inode or file size
        warning (or limit) level is reached, UNICOS will deliver a
        SIGINFO signal to this process.  By default, the signal is
        ignored by the process.  Since it was chosen to catch a SIGINFO
        signal, if one of these quotas is reached, the process is
        interrupted and the signal handling routine named "handler"
        is executed.  In "handler" the getinfo(2) request is made to
        request more detailed information from the kernel as to why
        the signal was sent (that is, what quota was exceeded).  If
        "handler" does not exit, then processing continues. */

}

void handler(int signo)
{
     long reason;

     printf("A signal of type %d (SIGINFO) was caught.  ", signo);
     printf("See following for reason.\n");
     if (getinfo(GI_SIGINFO, 0, (char *) &reason, sizeof(long)) == -1) {
          perror("getinfo failed");
          exit(1);
     }
     printf("The SIGINFO mask = %o (see getinfo.h for reason)\n", reason);
}
```

SEE ALSO

`quotactl`(2), `signal`(2)

NAME

`getjtab` – Gets the job table entry associated with a process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/jtab.h>

int getjtab (struct jtab *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getjtab` system call gets a copy of the job entry associated with the current process. It accepts the following arguments:

buf Points to the `jtab` structure. Information concerning the job is placed in this structure.

The `jtab` structure includes the following members (for a complete list, see `/usr/include/sys/jtab.h`):

```
int           j_jid;                 /* Job id of this entry */
int           j_uid;                 /* Job owner user-id */
int           j_ppid;                /* Process-id of job parent */
int           j_signal;              /* Signal notification for job parent */
int           j_nice;                /* Nice value of job */
time_t        j_cpulimit;            /* Max #of cpu clocks allowed */
long          j_cproclimit;          /* Max #of concurrent processes allowed */
long          j_memlimit;            /* Memory size limit (clicks) */
long          j_fsblklimit;          /* Max # of file system blocks */
                                     /* to consume */
long          j_sdslimit;            /* SDS limit in clicks rounded up */
                                     /* to minimum allocation unit; */
                                     /* CRAY Y-MP systems and */
                                     /* Cray MPP systems */
time_t        j_ucputime;            /* Total user cpu time for the job */
time_t        j_scputime;            /* Total system cpu time for the job */
long          j_nprocs;              /* Total number of processes active */
long          j_memuse;              /* Memory size in use by job (clicks) */
long          j_memhiwat;            /* Maximum memory ever used */
                                     /* by job (clicks) */
long          j_fsblkused;           /* # of file system blocks consumed */
long          j_sdsuse;              /* SDS in use; CRAY Y-MP systems */
                                     /* and Cray MPP systems */
unsigned char j_tapelimit [J_NTAPES]; /* Tape group limits; enforced */
                                     /* by tape  daemon */
```

RETURN VALUES

If `getjtab` completes successfully, the job ID is returned. A job ID of 0 indicates that the process is not part of a job. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getjtab` system call fails if the *buf* argument points to an address that was not valid, `EFAULT`.

EXAMPLES

This `getjtab` example shows how a process can retrieve all of the information about a job associated with it:

```
#include <sys/types.h>
#include <sys/jtab.h>

main()
{
     struct jtab jdata;
     int jid;

     if ((jid = getjtab(&jdata)) == 0) {
          fprintf(stderr, "This process is not part of a job!\n");
          exit(0);
     }
     else {
          if (jid == -1) {
               perror("getjtab failed");
               exit(1);
          }
     }

     /* information about the current job now available in jdata */
}
```

SEE ALSO

`fork`(2), `intro`(2), `killm`(2), `limit`(2), `nicem`(2), `setjob`(2), `signal`(2), `suspend`(2)

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

NAME

getlim – Obtains current resource limit information

SYNOPSIS

```
#include <errno.h>
#include <sys/category.h>
#include <sys/resource.h>

int getlim (int id, struct resclim *rptr);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The getlim system call gets resource limit information from the kernel based on the following arguments:

id Specifies the PID, SID, or UID corresponding to the resclim field resc_category. A 0 indicates the current PID, SID, or UID.

rptr Points to the resclim structure. The resclim structure pointed to by *rptr* includes the following members (for a complete list, see /usr/include/sys/resource.h):

```
struct resclim {
    int  resc_resource;          /* One of: L_CPU */
    int  resc_category;          /* One of: C_PROC, C_SESS, C_UID, C_SESSPROCS */
    int  resc_type;              /* One of: L_T_ABSOLUTE, L_T_HARD, L_T_SOFT */
    int  resc_action;            /* One of: L_A_TERMINATE, L_A_CHECKPOINT */
    long resc_used;              /* Current amount of resource used */
    long resc_value[R_NLIMTYPES];/* Current resource limit value */
                                 /* for each of: */
                                 /* L_T_ABSOLUTE, L_T_HARD, L_T_SOFT */
};
```

The resclim structure fields resc_resource and resc_category must be set in order to return limit values. The resc_resource field represents the resource to be queried. Currently, only CPU resources are supported; therefore, the value of resc_resource must be L_CPU. The resc_category identifies which category of resource to be queried. Acceptable values are: C_PROC, C_SESS, C_UID, and C_SESSPROCS. A short description follows:

Value	Description
C_PROC	Returns process limits
C_SESS	Returns session limits
C_UID	Returns user limits
C_SESSPROCS	Returns default process limits for the session

The resclim field resc_category determines whether the *id* argument is a PID, SID, or UID. The resc_category of C_SESSPROCS requires an SID.

If the call succeeds, getlim fills in the missing information in the resclim structure. This includes the following fields:

Field	Description
resc_action	Returns a value of L_A_TERMINATE or L_A_CHECKPOINT. This value determines whether when a hard limit is reached the process is checkpointed before termination.
resc_used	Returns the amount of resource currently accumulated at the time of the call. For L_CPU, this value is the amount of CPU clocks accumulated.
resc_value[R_NLIMTYPES]	Returns three values. The resc_value[L_T_ABSOLUTE] field is the absolute resource limit. The resc_value[L_T_HARD] field is the hard resource limit and resc_value[L_T_SOFT] is the soft resource limit. All CPU resource limits are returned in clocks.

NOTES

The active security label of the process must be greater than or equal to the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is allowed to override security label restrictions.
PRIV_POWNER	The process is considered the owner of every affected process.

If the PRIV_SU configuration option is enabled, the super user is considered the owner of every affected process. The super user is allowed to override security label restrictions.

RETURN VALUES

If getlim completes successfully, a value of 0 is returned, and the resclim structure is filled in with appropriate returned values. Otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The getlim system call fails and no information is updated in the resclim structure if one of the following conditions occurs:

Error Code	Description
EFAULT	The address specified for *rptr* was not valid.
EINVAL	One of the arguments contains a value that was not valid.

`EPERM`	The process does not own every affected process and does not have appropriate privilege.
`ESRCH`	No processes were found that matched the request.
`ESRCH`	The active security label of the process is not greater than or equal to the security label of every affected process, and the process does not have appropriate privilege.

SEE ALSO

`setlim`(2)

`nlimit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`nlimit`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

getmount – Returns information about the kernel mount table

SYNOPSIS

```
#include <sys/mount.h>
int getmount (struct mntentinfo *mountcopy, struct kmountinfo *mountinfo);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The getmount system call returns general information about the kernel mount table (number of mounted file systems, last time the kernel mount table has been changed, and so on) and information about each mounted file system (file system name, mount point, type, and options). The getmount system call returns all or only part of the information, depending on the value of the following arguments:

mountcopy Points to a set of mntentinfo structures that contains information about each file system.

mountinfo Points to the structure that contains the general information about the kernel mount table.

The mntentinfo structure is defined in the sys/mount.h include file, as follows:

```
struct mntentinfo {
     char *fname;          /* file system name */
     char *dir;            /* mount directory */
     char *opts;           /* mount options */
     char *type;           /* file system type */
};
```

The kmountinfo structure is defined in the sys/mount.h header file, as follows:

```
struct kmountinfo {
    int                nbent;      /* number of mounted file systems */
    long               lastchge;   /* last change in the mount table */
    int                needupdate; /* need to update memory space */
    struct mount       *rootenv;   /* root in the current environment */
    struct mountlength *len;       /* length of the character arrays */
};
```

If you need only general information about the kernel mount table, you can call getmount with the following arguments:

```
struct kmountinfo mountinfo;
getmount(NULL, &mountinfo);
```

In this case, only `nbent` and `lastchge` are valid in the `kmountinfo` structure.

If you need information about each mounted file system, use the `setmntent`(3C) and `getmntinfo`(3C) library routines. These routines use the `getmount` system call to access the kernel mount table. You should avoid using `getmount` directly. However, if it becomes necessary to use it, you must perform the following steps:

1. Allocate memory space for the set of `mountlength` structures `len`, declared as follows:

```
struct mntentinfo {
    int fslen;   /*file system length */
    int dirlen;  /* mount directory length */
    int optlen;  /* options length */
};
```

   Ensure that enough space is available for each entry in the kernel mount table.

2. Call `getmount` to get the general information about the kernel mount table, as follows:

```
getmount(NULL, &mountinfo);
```

   At this point, *mountinfo* should contain the number of mounted file systems and the space needed for the file system name, the mount point, and the options of each mounted file system.

3. Allocate memory space for the set of `mntentinfo` structures `mountcopy`.

4. Call the `getmount` system call by using a pointer to the allocated memory:

```
getmount(mountcopy, &mountinfo);
```

   The system call returns the requested information in the set of `mntentinfo` structures (`mountcopy`).

RETURN VALUES

If `getmount` completes successfully, 0 is returned; otherwise, –1 is returned, and `errno` is set to `EFAULT` to indicate the error.

EXAMPLES

The following example illustrates use of the `getmount` system call. The program gets information about each mounted file system from the kernel mount table and prints it.

```
#include <sys/types.h>
#include <sys/fstyp.h>
#include <sys/mount.h>

#include <stdio.h>
#include <unistd.h>

main()
   {
   int i, nb, nmnt;
   struct kmountinfo mountinfo= {0, 0, 0, NULL, NULL};
   struct mntentinfo *mountcopy;

   /* Allocate enough memory for the set of mountlength structures */
   nmnt = sysconf(_SC_CRAY_NMOUNT);
   mountinfo.len = (struct mountlength *)
                malloc(nmnt*sizeof(struct mountlength));

   /* get the general information about the kernel mount table */
   getmount(NULL, &mountinfo);

   /* allocate enough memory to get the information about each FS */
   nb = mountinfo.nbent;

   mountcopy = (struct mntentinfo *)malloc(nb*sizeof(struct mntentinfo));

   for (i = 0; i < nb; i++) {
            mountcopy[i].fname = (char *)malloc(mountinfo.len[i].fslen + 1);
            mountcopy[i].dir = (char *)malloc(mountinfo.len[i].dirlen + 1);
            mountcopy[i].opts = (char *)malloc(mountinfo.len[i].optlen + 1);
            mountcopy[i].type = (char *)malloc(FSTYPSZ + 1);
}

/* get the information about each mounted file system */
getmount(mountcopy, &mountinfo);

/* print it */
for (i = 0; i < nb; i++) {
        fprintf(stdout, "file system name: %s\n", mountcopy[i].fname);
        fprintf(stdout, "mount point: %s\n", mountcopy[i].dir);
        fprintf(stdout, "mount options: %s\n", mountcopy[i].opts);
        fprintf(stdout, "file system type: %s\n", mountcopy[i].type);
    }
}
```

FILES

`/usr/include/sys/mount.h` Contains C prototype for the `getmount` system call

SEE ALSO

`getmntinfo`(3C), `setmntent`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`mount`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`getpal` – Gets the privilege assignment list (PAL) and privilege sets of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int getpal (char *path, pal_t *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getpal` system call gets the privilege assignment list (PAL) and privilege sets of a file and returns the information in a buffer.

The `getpal` system call accepts the following arguments:

path Points to the file for which the PAL and privilege sets are retrieved.

buf Specifies the return buffer.

bufsize Indicates the maximum size of the buffer in bytes.

The calling process must have MAC read access to the file or have `PRIV_MAC_READ` in its effective privilege set.

RETURN VALUES

If `getpal` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getpal` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the *path* prefix denies search permission.
`EACCES`	The caller is denied MAC read access to the file.
`EFAULT`	The *buf* or *path* argument points outside the address space of the process.
`EINVAL`	The *bufsize* argument specifies an invalid value.
`ENAMETOOLONG`	The supplied file name is too long.
`ENOENT`	The specified file does not exist.

SEE ALSO

`fgetpal`(2), `fsetpal`(2), `setpal`(2)

NAME

`getpeername` – Gets name of connected peer

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

`int getpeername (int` *s*`, struct sockaddr *`*name*`, int *`*namelen*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getpeername` system call returns the name (*name*) of the peer connected to socket (*s*). You must initialize the *namelen* argument to indicate the amount of space to which *name* points. On return, *namelen* contains the actual number of bytes in the name returned. If the buffer that is provided is too small, the name is truncated.

The `getpeername` system call accepts the following arguments:

s Identifies the socket for which the address is desired.

name Points to the address of a `sockaddr` structure that receives the address of the remote socket connected to *s*.

namelen Points to an integer that receives the length of the address placed in *name*.

NOTES

If the `SOCKET_MAC` option is enabled, the active security label of the process must be greater than or equal to the security label of the socket. Note that `SOCKET_MAC` is part of TCP/IP configurable feature variables list in `uts/cf/Nmakefile`. For more information, see the `connect`(2) man page.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_MAC_READ`	The process is allowed to override the security label restrictions when the `SOCKET_MAC` option is enabled.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security label restrictions when the `SOCKET_MAC` option is enabled.

If *s* is an Internet-domain socket, the `sin_addr` and `sin_port` fields of *name* identify only the socket at the other end of the connection and not the remote process or remote user. Additional knowledge is required to interpret those fields. For example, if the `sin_addr` designates another UNICOS system, a `sin_port` value of less than 1024 indicates a connection with trusted software (for example, `rlogin`(1B)), which may include additional identity information in its protocol data stream. If it is necessary to identify the actual user associated with the socket, the communicating peers must agree in advance on a method, such as the sender placing its `sin_port` value in a protected file accessed through NFS (or other means) by the receiver.

Because no sender name information can be obtained from a UNIX-domain socket, the other end of the connection cannot be identified except to the extent that additional authentication techniques are used. Although there are no identity-based access controls that restrict use of `connect`(2) or `sendto`(2) for a UNIX-domain socket, such a socket can be created in a directory to which execute (search) access is restricted. This limits the ability of other processes to connect to the socket. Alternatively, the listening process could place a random value or secret password in a protected file and require that to be included in all messages it accepts; this ensures that only users with access to that file can send valid messages. For both Internet-domain and UNIX-domain, this authentication requires explicit action on the part of the receiver.

RETURN VALUES

If `getpeername` completes successfully, a value of 0 is returned; otherwise, –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getpeername` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	If the `SOCKET_MAC` option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
`EBADF`	Descriptor *s* is not valid.
`EFAULT`	Argument *name* points to memory that is not in a valid part of the process address space.
`ENOBUFS`	Insufficient system resources were available to perform the operation.
`ENOTCONN`	Socket is not connected.
`ENOTSOCK`	Descriptor *s* is not a socket.

FILES

`/usr/include/sys/socket.h`	Contains the header file for sockets
`/usr/include/sys/types.h`	Contains the header file for types

SEE ALSO

`bind`(2), `connect`(2), `getsockname`(2), `sendto`(2), `socket`(2)

`rlogin`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

getpermit, setpermit – Gets or sets user permissions

SYNOPSIS

```
#include <unistd.h>
#include <sys/perm.h>
#include <sys/category.h>

int getpermit (long *mask);

int setpermit (int cat, long *mask);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The getpermit system call allows a user program to determine the current permissions associated with the current process. The current permissions are returned in the word indicated by *mask*.

The setpermit system call allows a user program to set the permissions associated with the current process or job. Child processes inherit permissions from their parents. Any process can reduce its active permissions, but only an appropriately privileged process can increase the active permissions.

The getpermit and setpermit system calls accept the following arguments:

mask Specifies the word that contains the current permissions.

cat Determines whether the permissions go into effect for the current process (C_PROC) or the entire job (C_JOB).

The following is a list of possible permissions from the user database (UDB) (in file /usr/include/udb.h), descriptions, and possible permissions from the kernel (in file /usr/include/sys/perm.h). The UDB permission bit must be used when comparing to the udb structure, and kernel permissions must be used when using the system calls. For more complete descriptions of the permbits, see libudb(3C).

UDB Permissions	Description	Kernel Permissions
PERMBITS_ACCT	Accounting permission	PERM_ACCT
PERMBITS_ACCTID	Allows any account ID (newacct(1))	none
PERMBITS_ASKACID	Query for active acid	none
PERMBITS_BYPASSLABEL	Bypasses label tape processing	none
PERMBITS_CHOWN	Allows chown(2), chgrp(1), chmod(2)	PERM_CHOWN
PERMBITS_CHROOT	Allows chroot(2) permission	PERM_CHROOT
PERMBITS_DEDIC	Dedicate CPU permission	PERM_DEDIC

UDB Permissions	Description	Kernel Permissions
PERMBITS_DEVMAINT	Allows device diagnostic	PERM_DIAG
PERMBITS_GROUPADM	Group administrator	none
PERMBITS_GUARD	Driver DONUT guard mode	PERM_GUARD
PERMBITS_GUEST	Allows use of guest	PERM_GUEST
PERMBITS_GUESTADM	Guest administrator	PERM_GUESTADM
PERMBITS_ID	Allows ID changes	PERM_ID
PERMBITS_IPCPERSIST	Allows persistent IPC	PERM_IPCPERSIST
PERMBITS_MKNOD	Allows mknod(2) flag	PERM_MKNOD
PERMBITS_MLSMNT	Allows secure file system mount	Unused. This permbit is available to assign to user accounts, but it no longer grants special abilities.
PERMBITS_MOUNT	Allows mount(2)	PERM_MOUNT
PERMBITS_NICE	Allows nice(2) negative values	PERM_NICE
PERMBITS_NOBATCH	Disables batch logins	none
PERMBITS_NOIACTIVE	Disables interactive logins	none
PERMBITS_PLOCK	Allows plock(2)	PERM_PLOCK
PERMBITS_REALTIME	Allows real-time execution permission	PERM_RTIME
PERMBITS_RESLIM	Resource limits permission	PERM_RESC
PERMBITS_RESTRICTED	System login restricted (udbrstrict(8))	none
PERMBITS_SIGANY	Sends signals to anyone	PERM_SIG
PERMBITS_SUSPRES	Suspends/resumes outside job allowed	PERM_SUSPRES
PERMBITS_SYSPARAM	Sets system parameters permission	PERM_SYSP
PERMBITS_TAPEMANAGER	Allows special tape requests	PERM_TAPE
PERMBITS_WRUNLABEL	R/W unlabeled tapes	none
PERMBITS_YP	Yellow pages flag	none

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_RESOURCE	The process is allowed to increase its active permissions.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_RESLIM permbit is allowed to increase its active permissions.

RETURN VALUES

When getpermit or setpermit completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The getpermit or setpermit system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The *mask* argument points to memory that is not in a valid part of the process address space.
EPERM	The process does not have appropriate privilege to increase its active permissions.

EXAMPLES

This example shows how to use the getpermit system call to retrieve user permissions. The getpermit request retrieves the user permissions enabled for the current process and places them in the long integer named *mask*.

A bit conversion table in header file udb.h simplifies the interpretation of these permissions, which are represented as single bits in the return value placed in *mask*. In this example, the table is used to convert each permission bit represented in *mask* to a meaningful character string for display.

```
/* These must be defined to use the udb conversion tables and
   their definitions must precede the "#include <udb.h>" */

#define UDB_BIT_CONVERSION  1
#define UDB_BIT_TABLE       1

#include <sys/types.h>
#include <udb.h>
#include <unistd.h>

main()
{
     long mask, permit, i;
     int j;

     if (getpermit(&mask) == -1) {
          perror("getpermit failed");
          exit(1);
     }

     if (mask == 0) {
          printf("\nNo permits currently enabled for this process!\n\n");
     }
     else {
```

```
                printf("\nCurrent permits enabled for this process:\n");
                for (i = 1L; i > 0; i <<= 1) {
                     if (permit = mask & i) {
                          for (j = 0; perm_def[j].pmask != 0; j++) {
                               if (perm_def[j].pmask == permit) {
                                    printf("     %s\n", perm_def[j].pname);
                                    break;
                               }
                          }
                     }
                }
           }
      }
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `getpermit` and `setpermit` system calls

SEE ALSO

`chmod`(2), `mknod`(2), `mount`(2), `nice`(2), `plock`(2)

`chgrp`(1), `login`(1), `newacct`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`libudb`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`cpu`(4), `udb`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`acct`(8), `csanqs`(8), `udbrstrict`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`getpid`, `getpgrp`, `getppid`, `_getlwpid`, `_getlwppid`, `newgetpid`, `newgetppid` – Gets process, process group, or parent process IDs

SYNOPSIS

All Cray Research systems:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid (void);

pid_t getpgrp (void);

pid_t getppid (void);
```

Cray PVP systems:

```
#include <sys/types.h>
#include <sys/unistd.h>

pid_t _getlwpid (void);

pid_t _getlwppid (void);

pid_t newgetpid (void);

pid_t newgetppid (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `getpid`, `getpgrp`, `getppid`)

DESCRIPTION

The following system calls obtain the process IDs of the calling process:

- The `getpid` and `newgetpid` system calls return the process ID.
- The `getpgrp` system call returns the process group ID.
- The `getppid` and `newgetppid` system calls return the parent process ID.
- The `_getlwpid` system call returns the light-weight process ID.
- The `_getlwppid` system call returns the light-weight process ID of the process that created the caller's process.

NOTES

The process ID interfaces have changed in UNICOS 9.0 to support a new multitasking model. Whereas previous UNICOS releases support multiple process IDs in a multitasking group, the new model supports a single process ID. In this new model, what were previously called process IDs are called *light-weight process IDs*.

The `_getlwpid` and `_getlwppid` system calls are not recommended for general use; they are provided for use by system software.

The `newgetpid` and `newgetppid` system calls will be removed in a subsequent UNICOS release.

FORTRAN EXTENSIONS

The `getpid` system call can be called from Fortran as a function:

```
INTEGER GETPID, I
I = GETPID ( )
```

The `getpgrp` system call can be called from Fortran as a function:

```
INTEGER GETPGRP, I
I = GETPGRP ( )
```

The `getppid` system call can be called from Fortran as a function:

```
INTEGER GETPPID, I
I = GETPPID ( )
```

The `newgetpid` system call can be called from Fortran as a function (on all systems except Cray MPP systems):

```
INTEGER NEWGETPID, I
I = NEWGETPID ( )
```

The `newgetppid` system call can be called from Fortran as a function (on all systems except Cray MPP systems):

```
INTEGER NEWGETPPID, I
I = NEWGETPPID ( )
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `getpid`, `getpgrp`, `getppid`, `_getlwpid`, `_getlwppid`, `newgetpid`, and `newgetppid` system calls

SEE ALSO

`exec`(2), `fork`(2), `intro`(2), `setpgrp`(2), `signal`(2)

NAME

`getppriv` – Gets the privilege state of the calling process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>
```

`int getppriv (priv_proc_t *`*buf*`, int `*bufsize*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getppriv` system call gets the privilege state of the calling process and places it in a buffer.

The `getppriv` system call accepts the following arguments:

buf Points to the return buffer.

bufsize Indicates the maximum size of the buffer in bytes.

RETURN VALUES

If `getppriv` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getppriv` system call fails if the following error condition occurs:

Error Code	Description
`EFAULT`	The *buf* argument points outside the address space of the process.

SEE ALSO

`setppriv`(2)

NAME

`getsectab` – Gets security names and associated values

SYNOPSIS

```
#include <sys/sectab.h>
int getsectab (int type, struct sectab *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getsectab` system call returns security names and associated values.

The `getsectab` system call accepts the following arguments:

type Specifies the type of names and values to be returned. The following are valid values:

Value	Description
`0`	Returns security compartment names and bit mask values.
`1`	Returns permission names and bit mask values.
`2`	Returns category names and bit mask values.
`3`	Returns security flags and bit mask values.
`4`	Returns security level names and numbers.
`6`	Returns privilege names and bit mask values.

buf Points to the `sectab` structure in which the values are returned. A list of names (maximum 64) and a list of values (–1 terminated) are returned.

RETURN VALUES

If `getsectab` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getsectab` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The *buf* argument points outside the process address space.
`EINVAL`	The *type* argument is less than 0 or greater than 5.

EXAMPLES

The following example shows how to use the `getsectab` system call to retrieve all of the security names and their associated values. For each type of name to be retrieved, `getsectab` is called once. The names and associated values are then displayed on `stdout`.

```
#include <sys/sectab.h>
#include <string.h>

main()
{
     static char *names[] = {"Compartment", "Permission", "Integrity category",
                             "Flag", "Security level", "Integrity class",
                             "Privilege"};
     struct sectab sectab;
     int i, j;

     for (i = MINTAB; i <= MAXTAB; i++) {
          if (getsectab(i, &sectab) == -1) {
              perror("getsectab failed");
              exit(1);
          }

          printf("%s names and values(octal):\n\n", names[i]);
          for (j = 0; j < MAXNAMES; j++) {
              if (strlen(sectab.tb_name[j]) == 0 || sectab.tb_num[j] == -1) {
                   continue;       /* ignore null table entries */
              }
              printf("%-25s      %21o\n", sectab.tb_name[j], sectab.tb_num[j]);
          }
          printf("\n");
     }
}
```

FILES

`/usr/include/sys/sectab.h`	Defines structure in which to return security name and value information

SEE ALSO

`secbits`(3C), `secnames`(3C), `secnums`(3C), `secwords`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`sectab`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getsockname – Gets socket name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

int getsockname (int *s*, struct sockaddr **name*, int **namelen*);

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The getsockname system call returns the current name (*name*) of the specified socket (*s*). You must initialize the *namelen* argument to indicate the amount of space to which *name* points. On return, *namelen* contains the actual number of bytes in the name returned.

The getsockname system call accepts the following arguments:

s	Identifies the socket.
name	Points to the current name of the socket.
namelen	Points to the size of the *name* array.

NOTES

If the SOCKET_MAC option is enabled, the active security label of the process must be greater than or equal to the security label of the socket. Note that SOCKET_MAC is part of TCP/IP configurable feature variables list in uts/cf/Nmakefile. For more information, see the connect(2) man page.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_MAC_READ	The process is allowed to override the security label restrictions when the SOCKET_MAC option is enabled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions when the SOCKET_MAC option is enabled.

RETURN VALUES

If getsockname completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The `getsockname` system call fails if one of the following conditions occurs:

Error Code	**Description**
EACCES	If the `SOCKET_MAC` option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EBADF	Descriptor *s* is not valid.
EFAULT	Argument *name* or *namelen* points to memory that is not in a valid part of the process address space.
ENOBUFS	Insufficient system resources were available to perform the operation.
ENOTSOCK	Descriptor *s* is not a socket.

FILES

`/usr/include/sys/socket.h`	Contains the header file for sockets
`/usr/include/sys/types.h`	Contains the header file for types

SEE ALSO

`bind`(2), `socket`(2)

NAME

`getsockopt`, `setsockopt` – Gets or sets options on sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

`int getsockopt (int` *s*`, int` *level*`, int` *optname*`, char *`*optval*`, int *`*optlen*`);`

`int setsockopt (int` *s*`, int` *level*`, int` *optname*`, char *`*optval*`, int` *optlen*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getsockopt` and `setsockopt` system calls manipulate options associated with a socket. Options can exist at multiple protocol levels; they are always present at the uppermost (socket) level.

The *getsockopt* system call allows an application to request information about a socket. The `setsockopt` system call allows an application to manipulate options associated with a socket.

The `getsockopt`(2) and `setsockopt`(2) accept the following arguments:

s Specifies the descriptor for the socket.

level Specifies the level at which the option resides. To manipulate options at the socket level, specify *level* as `SOL_SOCKET`. To manipulate options at any other level, supply the protocol number of the protocol that controls the option (for example, to indicate that an option is interpreted by the Transmission Control Protocol (TCP) protocol, set *level* to the protocol number of TCP). For more information, see the `getprot`(3C) man page.

optname Specifies the name of the option to examine or retrieve. For the list of available options, see the Options subsection.

optval For `getsockopt`(2), identifies a buffer in which the current values for the requested options are returned.

For `setsockopt`(2), identifies a buffer in which the new values for the specified options are retrieved.

If no option value is supplied or returned, *optval* can be supplied as 0.

optlen For `getsockopt`(2), initially contains the size of the buffer to which *optval* points and is modified on return to indicate the actual size of the value returned. It is a value-result argument.

For `setsockopt`(2), specifies the length of the new values residing in the buffer to which *optval* points.

The *optname* argument and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The `sys/socket.h` include file contains definitions for socket-level options (see `socket`(2)). Options at other protocol levels vary in format and name (see the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014).

Most socket-level options take an `int` type value for *optval*. For `setsockopt`, the value must be nonzero to enable a Boolean option, or 0 if the option will be disabled. The `SO_LINGER` option uses a `struct linger` value, defined in `sys/socket.h`, which specifies the desired state of the option and the linger interval.

Options

The following options are recognized at the socket level. Except as noted, each may be examined with `getsockopt` and set with `setsockopt`.

Option	**Description**
`SO_BROADCAST`	Toggles permission to transmit broadcast messages.
`SO_DEBUG`	Toggles recording of debugging information. This option enables debugging in the underlying protocol modules.
`SO_REUSEADDR`	Toggles local address reuse. This option indicates that the rules used in validating addresses supplied in a `bind`(2) system call should allow reuse of local addresses.
`SO_REUSEPORT`	Allows multiple processes to completely duplicate bindings, if all processes set `SO_REUSEPORT` before binding the port. Every process must specify this option, including the first process to use the port. This option permits multiple instances of a program to receive UDP/IP multicast or to broadcast datagrams for the bound port.
`SO_OWNPORT`	Allows a process to bind to a port and prevent other applications from binding to the same port. This option has been improved to provide better defined addresses.
`SO_KEEPALIVE`	Toggles keep connections alive. This option enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken, and processes using the socket are notified through a `SIGPIPE` signal.
`SO_DONTROUTE`	Toggles routing bypass for outgoing messages. This option indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER	Lingers on close if data is present. This option controls the action taken when unsent messages are queued on socket, and a close(2) system call is performed. If the socket promises reliable delivery of data, and SO_LINGER is set, the system blocks the process on the close attempt until it can transmit the data or until it decides it cannot deliver the information (a time-out period, termed the *linger interval*, is specified in the setsockopt call when SO_LINGER is requested). If SO_LINGER is disabled, and a close(2) system call is issued, the system processes the close operation in a manner that allows the process to continue as quickly as possible.
SO_SNDBUF	Sets buffer size for output.
SO_RCVBUF	Sets buffer size for input. SO_SNDBUF and SO_RCVBUF are options that can be used to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The system places limits on these values. There is a system-wide maximum size for an individual socket buffer. The system silently sets the socket buffer size to the system-wide limit if the request exceeds the limit. This limit is configurable through the netvar(8) command. There is also a per-session cumulative limit for all of the sockets used by a session. The setsockopt call returns ELIMIT and leaves the current value unchanged if the new size would exceed the session limit. This limit is configurable through the user's udb entry.
SO_TYPE	Gets the type of the socket (get only). This option is used only with getsockopt. It returns the type of the socket (for example, SOCK_STREAM); it is useful for servers that inherit sockets on startup.
SO_ERROR	Gets and clears error on the socket (get only). This option is used only with getsockopt. It returns any pending error on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets (type SOCK_DGRAM) or for other asynchronous errors. Values for SO_ERROR correspond to values for errno.
SO_USELOOPBACK	Bypasses hardware when possible.
SO_SNDLOWAT	Sends low-water mark.
SO_RCVLOWAT	Receives low-water mark.
SO_SNDTIMEO	Sends time-out.
SO_RCVTIMEO	Receives time-out.
SO_DOBINLINE	Leaves received out-of-band data in line.

NOTES

If the SOCKET_MAC option is enabled, to set options on a socket, the active security label of the process must equal the security label of the socket. The SOCKET_MAC option is part of TCP/IP configurable feature variables list in uts/cf/Nmakefile. For more information, see the connect(2) man page.

If the SOCKET_MAC option is enabled, to get options on a socket, the active security label of the process must be greater than or equal to the security label of the socket.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	When the SOCKET_MAC option is enabled, the process is allowed to override the security label restrictions when getting options of a socket.
PRIV_MAC_WRITE	When the SOCKET_MAC option is enabled, the process is allowed to override the security label restrictions when setting options on a socket.

If the PRIV_SU configuration option is enabled, the super user is allowed to override security label requirements when the SOCKET_MAC option is enabled.

RETURN VALUES

If getsockopt or setsockopt completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The getsockopt or setsockopt system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	If the SOCKET_MAC option is enabled, the process does not meet the security label requirements and does not have the appropriate privilege.
EBADF	The *s* descriptor is not valid.
EFAULT	The options are not in a valid part of the process address space.
EINVAL	The option or level specified is not valid.
ELIMIT	The request would exceed the session’s limit.
ENOBUFS	No buffer space is available.
ENOPROTOOPT	The option is unknown; therefore, it has not been obtained (for getsockopt) or set (for setsockopt).
ENOTSOCK	The *s* descriptor is not a socket.

FILES

`/usr/include/sys/socket.h`	Contains the header file for sockets
`/usr/include/sys/types.h`	Contains the header file for types

SEE ALSO

`bind`(2), `close`(2), `socket`(2)

`getprot`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`ip`(4P), `tcp`(4P) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`netvar`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

NAME

getsysv – Gets security attributes

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysv.h>

int getsysv (struct sysv *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getsysv` system call returns security attributes. Any process may issue a `getsysv` request.

The `getsysv` system call accepts the following arguments:

buf Points to a `sysv` structure in which information is returned.

bufsize Specifies the size of the `sysv` structure in bytes.

The `sysv` structure includes the following members:

```
    short   sy_minlvl;              /* minimum security level */
    short   sy_maxlvl;              /* maximum security level */
    long    sy_valcmp;              /* authorized compartments */
    int     sy_slgbufsize;          /* size (in bytes) of /dev/slog */
    char    sy_secure_console[24];  /* secure console: administrator (not used) */
    char    sy_admin_console[24];   /* default console: administrator
                                       (must be /dev/console) */
    char    sy_oper_console[24];    /* secure console: operator (not used) */
    int     sy_spare1;              /* filler */
    uint    sy_dev_enforce   :  1,  /* Device labeling enforcement */
    uint    sy_spare2        : 22,  /* filler */
            sy_ranpass_flag  :  1,  /* machine passwords enabled if set */
            sy_ranpass_min   :  4,  /* machine passwords minimum size */
            sy_ranpass_max   :  4,  /* machine passwords maximum size */
            sy_netw_options  : 32;  /* network security options */
    int     sy_overwrite_count;     /* declassify disk overwrite count */
    int     sy_declassify_pattern;  /* declassify disk write pattern */
    int     sy_sanitize_pattern;    /* scrub disk write pattern */
    int     sy_maxlogs;             /* max login attempts before disable */
    int     sy_logdelay;            /* delay (seconds) between failed
                                       login attempts */
    int     sy_disable_time;        /* duration (seconds) for which a user
                                       is disabled for exceeding maxlogs */
    int     sy_delay_mult    : 1;   /* multiply sy_logdelay by the
                                       number of successive failed
                                       login attempts to calculate
                                       the delay time (in seconds) */
    int     sy_slg_state     : 1;   /* security log state on/off */
    int     sy_slg_discv     : 1;   /* log discretionary violations */
    int     sy_slg_mandv     : 1;   /* log mandatory violations */
    int     sy_slg_netwv     : 1;   /* log network violations */
    int     sy_slg_mkdirv    : 1;   /* log mkdir violations */
    int     sy_slg_rmdirv    : 1;   /* log rmdir violations */
    int     sy_slg_linkv     : 1;   /* log link violations */
    int     sy_slg_all_rm    : 1;   /* log all rm requests */
    int     sy_slg_removev   : 1;   /* log rm violations */
    int     sy_slg_all_nami  : 1;   /* log all nami requests */
    int     sy_slg_all_valid : 1;   /* log all requests */
    int     sy_slg_all_netw  : 1;   /* log all network requests */
    int     sy_slg_physio_err: 1;   /* log physical I/O errors */
    int     sy_slg_path_track: 1;   /* track pathname of all entries */
    int     sy_sec_remote    : 1;   /* tcp/ip mand. access control */
    int     sy_sec_scrub     : 1;   /* scrub data blocks on delete */
    int     sy_nfs_export    : 1;   /* export secure fs via nfs */
```

```
int     sy_nfs_remote    : 1;   /* rw remote fs via nfs */
int     sy_oldtfm        : 1;   /* old style tfm is possible (not used) */
int     sy_stat_restrict : 1;   /* restrict (sec)stat by level */
int     sy_declsfy_disk  : 1;   /* enable declsfy disk action */
int     sy_slg_sulog     : 1;   /* log all su(1) attempts */
int     sy_console_msg   : 1;   /* send a message to the console when
                                /* MAXLOGS has been exceeded */
int     sy_disable_acct  : 1;   /* disable account when MAXLOGS */
                                /* exceeded */
int     sy_slg_filexfr   : 1;   /* log all file transfers */
int     sy_slg_nfs       : 1;   /* log CNFS activity */
int     sy_slg_config    : 1;   /* log UNICOS configuration changes */
int     sy_slg_jstart    : 1;   /* log start of job */
int     sy_slg_jend      : 1;   /* log end of job */
int     sy_slg_netcf     : 1;   /* log network configuration changes */
int     sy_slg_suid      : 1;   /* log suid requests */
int     sy_slg_user      : 1;   /* log user name & password for failed
                                   logins */
int     sy_slg_nqscf     : 1;   /* log NQS database changes */
int     sy_slg_nqs       : 1;   /* log NQS activity */
int     sy_slg_trust     : 1;   /* log trusted process activity */
int     sy_mac_command   : 1;   /* perform MAC checks in commands */
int     sy_forced_socket : 1;   /* ON: syslogd uses sockets only */
int     sy_slg_priv      : 1;   /* log use of privilege */
int     sy_audit_chng    : 1;   /* auditing criteria change flag */
int     sy_slg_audit     : 1;   /* log auditing criteria change */
int     sy_slg_chdir     : 1;   /* log chdir requests */
int     sy_slg_crl       : 1;   /* log Cray REEL librarian activity */
int     sy_slg_ipnet     : 1;   /* log IP layer activity */
int     sy_slg_oper      : 1;   /* log operator actions */
int     sy_slg_secsys    : 1;   /* log non_inode security syscalls */
int     sy_slg_shutdown  : 1;   /* log system shutdown */
int     sy_slg_startup   : 1;   /* log system startup */
int     sy_slg_tape      : 1;   /* log tape activity */
int     sy_slg_tchg      : 1;   /* log system time change */
int     sy_slg_dac       : 1;   /* log DAC changes */
int     sy_slg_privileged: 1;   /* privileged to change auditing */
int     sy_fsetid_restrict:1;   /* restrict setuid/setgid file mgmt */
int     sy_secure_pipe   : 1;   /* MAC restricted pipes? */
int     sy_spare3        : 9;   /* filler - use as needed */
int     sy_slg_maxsize;         /* maximum size of security log */
char    sy_slg_dir[PATH_MAX1+1];/* path for active security log */
char    sy_slg_file[PATH_MAX+1];/* filename for active log */
char    sy_slg_fprefix[PATH_MAX+1];/* prefix for retired log */
```

NOTES

The `getsysv` requests are not recorded in the security log.

If *bufsize* is greater than 0, but less than the defined size of a `sysv` structure, the user's buffer is filled in with the amount of data that fits and a successful status is returned.

RETURN VALUES

If `getsysv` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getsysv` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The *buf* argument points outside the process address space.
`EINVAL`	The *bufsize* argument is less than 0. If *bufsize* is greater than the size of the `sysv` structure, *bufsize* is bounded silently by the actual size.

EXAMPLES

The following example shows how to use the `getsysv` system call to retrieve security attributes.

Special handling of the bit status fields in the `sysv` structure is included.

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysv.h>

main()
{
     struct sysv buf;
     static char *answ[] = {"No", "Yes"};

     if (getsysv(&buf, sizeof(buf)) == -1)  {
          perror("getsysv failed");
          exit(1);
     }

     printf("System minimum security level = %d\n", buf.sy_minlvl);
     printf("System maximum security level = %d\n", buf.sy_maxlvl);
     printf("System authorized compartments= %o\n", buf.sy_valcmp);
     printf("Security log size (/dev/dslog)= %d\n", buf.sy_slgbufsize);

     /* The fields in the sysv structure starting with sy_delay_mult
        are all bit status fields with a 0 value meaning NO (not
        enabled) and a 1 meaning YES (is enabled).  The following
        statements print the status of some of these fields. */

     printf("Log discretionary violations  = %s\n", answ[buf.sy_slg_discv]);
     printf("Log mandatory violations      = %s\n", answ[buf.sy_slg_mandv]);
     printf("Log network violations        = %s\n", answ[buf.sy_slg_netwv]);
     printf("Log mkdir violations          = %s\n", answ[buf.sy_slg_mkdirv]);
}
```

FILES

`/usr/include/sys/param.h`	Defines configuration files
`/usr/include/sys/sysv.h`	Defines structure for system security values
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`setsysv`(2)

`spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`gettimeofday`, `settimeofday` – Gets or sets date and time

SYNOPSIS

```
#include <sys/time.h>
int gettimeofday (struct timeval *tp, struct timezone *tzp);
int settimeofday (struct timeval *tp, struct timezone *tzp);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `gettimeofday` system call gets the system's notion of the current Greenwich time, to microsecond accuracy. The `settimeofday` system call sets this time. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The accuracy of the system clock is hardware dependent, using the real-time clock.

The `gettimeofday` and `settimeofday` system calls accept the following arguments:

tp Points to the `timeval` structure.

tzp Points to the `timezone` structure.

The structures to which *tp* and *tzp* point are defined in the `sys/time.h` file, as follows:

```
struct timeval {
    long   tv_sec;           /* seconds since Jan. 1, 1970 */
    long   tv_usec;          /* and microseconds */
};

struct timezone {
    int    tz_minuteswest;   /* of Greenwich */
    int    tz_dsttime;       /* type of dst correction to apply */
};
```

The `timezone` structure indicates the local time zone (measured in minutes of time westward from Greenwich) and a flag. A nonzero flag indicates that daylight saving time applies locally during the appropriate part of the year.

If *tzp* is a zero pointer, the time zone information is not returned or set.

An extended kernel timezone structure, `kn_timezone`, is defined in the `sys/time.h` file. The `gettimeofday` and `settimeofday` system calls may be used to set and retrieve the contents of this structure when *tp* points to a `timeval` structure that contains `-1` in the `tv_sec` field and the flag value `TZ_MAGIC` in the `tv_usec` field. The extended kernel timezone structure is copied to or from the location pointed to by *tzp*. In this case, the value of the time is returned by `gettimeofday`, but is not modified by `settimeofday`.

Only a process with appropriate privilege can set the time of day or time zones.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_TIME`	The process is allowed to set the time.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_SYSPARAM` permbit is allowed to set the time.

The *tzp* argument is compatible only with 4.3 BSD source code. Except in the special case of the extended kernel timezone request described above, the time zone information is silently ignored on a `settimeofday` request and always returns zeros on a `gettimeofday` request. Time zone information must always be obtained with the `ctime`(3C) library routines that honor the `TZ` environment variable.

RETURN VALUES

If `gettimeofday` or `settimeofday` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `gettimeofday` or `settimeofday` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	An argument address references memory that is not valid.
`EINVAL`	The value specified in `tv_usec` is out-of-range. Valid values are greater or equal to 0 and less than 1 million.
`EPERM`	The process does not have appropriate privilege to set the time.

EXAMPLES

The following example shows how to use the `gettimeofday` system call to obtain the time of day accurate to the microsecond:

```
#include <sys/time.h>

main()
{
     struct timeval tp;
     struct timezone tzp;

     if (gettimeofday(&tp, &tzp) == -1) {
          perror("gettimeofday failed");
          exit(1);
     }

     /* The system time to microsecond accuracy is now available
        in the timeval structure (tp).  Field tp.tv_sec contains
        the number of seconds since January  1, 1970, while
        field tp.tv_usec contains the number of additional
        microseconds. */
}
```

SEE ALSO

`time`(2)

`date`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ctime`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`getuid`, `geteuid`, `getgid`, `getegid` – Gets real user, effective user, real group, or effective group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid (void);

uid_t geteuid (void);

gid_t getgid (void);

gid_t getegid (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `getuid`, `geteuid`, `getgid`)

DESCRIPTION

The following system calls obtain the user and group IDs of the calling process:

- The `getuid` system call returns the real user ID.
- The `geteuid` system call returns the effective user ID.
- The `getgid` system call returns the real group ID.
- The `getegid` system call returns the effective group ID.

FORTRAN EXTENSIONS

The `getuid` system call can be called from Fortran as a function:

```
INTEGER GETUID, I
I = GETUID ( )
```

The `geteuid` system call can be called from Fortran as a function:

```
INTEGER GETEUID, I
I = GETEUID ( )
```

The `getgid` system call can be called from Fortran as a function:

```
INTEGER GETGID, I
I = GETGID ( )
```

The `getegid` system call can be called from Fortran as a function:

```
INTEGER GETEGID, I
I = GETEGID ( )
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `getuid`, `geteuid`, `getgid`, and `getegid` system calls

SEE ALSO

`intro`(2), `setuid`(2)

NAME

`getusrv` – Gets security validation attributes of the process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/usrv.h>

int getusrv (struct usrv *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `getusrv` system call obtains security validation information for the calling process. It accepts the following argument:

buf Points to a `usrv` structure in which the information is returned.

A `usrv` structure includes the following members:

```
short   sv_minlvl;      /* minimum security level */
short   sv_maxlvl;      /* maximum security level */
long    sv_valcmp;      /* authorized compartments */
long    sv_savcmp;      /* TFM_EXEC command saved compartments (not used) */
long    sv_actcmp;      /* active compartments */
short   sv_permit;      /* permissions */
short   sv_actlvl;      /* active security level */
short   sv_savlvl;      /* TFM_EXEC saved security level (not used) */
short   sv_intcls;      /* active integrity class (not used) */
short   sv_maxcls;      /* maximum integrity class (not used) */
long    sv_intcat;      /* active categories */
long    sv_valcat;      /* authorized categories */
struct  {               /* saved integrity parameters over TFM_EXEC
                           (not used) */
        int     actcls  :32;    /* integrity class before TFM_EXEC
                                   (not used) */
        int     actcat  :32;    /* active category before TFM_EXEC
                                   (not used) */
} sv_savint;
int     sv_audit_off    :1;     /* audit on/off flag */
int     sv_audit_chng   :1;     /* audit change flag */
```

NOTES

The `getusrv` requests are not recorded in the security log.

RETURN VALUES

If `getusrv` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `getusrv` system call fails if the following error condition occurs:

Error Code	**Description**
`EFAULT`	The *buf* argument points outside the address space of the process.

EXAMPLES

The following example shows how to use the `getusrv` system call to retrieve the security attributes for the calling process. This program only displays the user's special permissions. A security administrator (`secadm`) or a trusted process is allowed to see the `usrtrap` flag; this information is not displayed to nonprivileged users.

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/usrv.h>

main()
{
     struct usrv buf;

     if (getusrv(&buf) == -1) {
          perror("getusrv failed");
          exit(1);
     }

     /* Security attributes for the calling process now available in the
        usrv structure named buf. */

     printf("My permissions = %o or interpreted as follows:\n", buf.sv_permit);
     if (PERMIT_SUIDGID & buf.sv_permit) printf ("   set-UID or set-GID\n");
     if (PERMIT_USRTRAP & buf.sv_permit) printf ("   user trap mode set\n");
}
```

FILES

`/usr/include/sys/usrv.h`	Contains C prototype for the `getusrv` system call

SEE ALSO

`setucat`(2), `setucmp`(2), `setulvl`(2), `setusrv`(2)

`setucat`(1), `setucmp`(1), `setulvl`(1), `spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`guestctl` – Controls and reports the status of major guest system functions

SYNOPSIS

```
#include <sys/guest.h>
#include <sys/guestctl.h>

int guestctl (struct gctl *gct);
```

IMPLEMENTATION

Cray PVP systems except CRAY J90 series and CRAY EL series

DESCRIPTION

The `guestctl` (guest control) system call interacts with a kernel to provide guest system status and to support the following requests:

```
#define GCTL_LOCK       1       /* get lock                     */
#define GCTL_UNLOCK     2       /* release lock                 */
#define GCTL_START      3       /* start guest                  */
#define GCTL_STOP       4       /* stop guest                   */
#define GCTL_STAT       5       /* return status                */
#define GCTL_CHANGE     6       /* change CPU allocation        */
#define GCTL_RESID      7       /* reserve id and memory        */
#define GCTL_LOAD       8       /* load guest kernel in memory  */
#define GCTL_FREEZE     9       /* inhibit guest user CPU usage */
#define GCTL_THAW       10      /* permit guest user CPU usage  */
#define GCTL_RELMEM     11      /* release guest memory         */
#define GCTL_RUNLVL     12      /* set system run level (GUEST) */
#define GCTL_RESUME     13      /* resume execution after breakpoint */
#define GCTL_GLOAD      14      /* load data into guest memory  */
```

A `gctl` structure includes the following members:

```
int     gs_id;                  /* kernel id                       */
char    *gs_start;              /* start of kernel in buffer       */
int     gs_ksize;               /* size of kernel in bytes         */
int     gs_psize;               /* size of param file in bytes     */
int     gs_mem;                 /* memory to assign/release        */
int     gs_ttyreq;              /* number of ttys requested        */
char    gs_owner[GS_OWNRSZ];    /* ASCII name of owner             */
int     gs_req;                 /* guestctl request                */
int     gs_status;              /* guestctl reply status           */
int     gs_maxguests;           /* configured number of guests     */
int     gs_grtcnt;              /* maximum # of grt entries        */
int     gs_trace;               /* extended trace flag             */
int     gs_runlvl;              /* system run level                */
int     gs_lockpid;             /* pid of lock owner               */
struct  gdesc gs_gdesc;         /* gdesc buffer                    */
uint    gs_spres  : 1,          /* != 0 if sched params present */
        unused    : 31,
        gs_cpuhog : 8,          /* CPU hog threshold               */
        gs_schint : 8,          /* CPU scheduling interval         */
        gs_winsiz : 8,          /* CPU scheduling window size      */
        gs_minres : 8;          /* minimum CPU residency time      */
word    *gs_gldaddr;            /* guest memory load addr (words)*/
int     gs_gldsize;             /* load size in bytes               */
long    gs_spares[7];           /* unused                          */
struct  gstt gstat[GS_maxguests+1];  /* guest status structure  */
```

A `gstt` structure (embedded in the `gct`) includes the following members:

```
char    gs_name[GS_NAMESZ];     /* system name                     */
long    gs_ttys;                /* assigned ttys                   */
int     gs_csim;                /* nonzero if csim                 */
int     gs_cpratio;             /* CPU ratio                       */
word    *gs_kba;                /* kernel fwa                      */
word    *gs_kla;                /* kernel lwa                      */
int     gs_memsize;             /* memory size                     */
char    gs_owner[GS_OWNRSZ];    /* ASCII name of owner             */
word    *gs_uba;                /* user fwa                        */
word    *gs_ula;                /* user lwa                        */
int     gs_stt;                 /* system status                   */
int     gs_frozen;              /* system frozen flag              */
int     gs_runlvl;              /* system run level                */
int     gs_halt;                /* halt host if guest panics       */
char    gs_panic[GS_PANICBUF];  /* panic buffer contents           */
int     gs_dedclus;             /* use dedicated system cluster */
long    gs_rval;                /* pkt validation disable flags */
long    gs_rovl;                /* res. overlap disable flags      */
long    gs_pkthdl;              /* packet handler flags            */
int     gs_memhole;             /* associated memory hole          */
int     gs_pktvallev;           /* IOS packet validation level  */
uint    gs_dedic  : 1,          /* dedicate requested CPUs         */
        gs_cpureq : 63;         /* number of CPUs requested        */
long    gs_spares[3];           /* unused                          */
```

The status request (`GCTL_STAT`) will return information about the host and any active guest in the `gstat` array of `gstt` structures. The first entry (0) is that of the host. Subsequent entries (0 through `GS_maxguests`) are valid if the allocated memory (`gs_memsize`) is greater than 0. The current status of each valid system entry is returned in `gs_stt` and can be printed using `gstatus[gct->gstat[id].gs_stt]`. Status values include:

```
#define GSS_NONE        0       /* no status                       */
#define GSS_RSRV        1       /* reserving memory & id           */
#define GSS_LOAD        2       /* loading kernel                  */
#define GSS_STRT        3       /* starting guest kernel           */
#define GSS_EXEC        4       /* guest kernel is executing       */
#define GSS_STOP        5       /* stopped normally                */
#define GSS_PANC        6       /* stopped due to panic            */
#define GSS_OBST        7       /* stopped but cpu not returned */
#define GSS_GLOAD       8       /* generic guest memory load       */
```

The lock/unlock (`GCTL_LOCK/GCTL_UNLOCK`) sequence should bracket all of the major `guestctl` requests. This prevents other users from making conflicting changes to the guest control structures. If the lock is already held (`gs_status == EGS_LOCKED`), the process id of the lock owner will be returned in `gs_lockpid`.

A guest system start requires the following five `guestctl` requests:

1. Lock the guest control structure (`GCTL_LOCK`)
2. Reserve a guest id and guest memory (`GCTL_RESID`)*

   Required Fields:

   `gs_mem` (size in words of requested guest memory)
   `gs_ttyreq` (number of desired OWS tty connections)

   Optional Fields:

   `gs_owner` (string representing the system owner)

   Returned if successful:

   `gs_id > 0`
   `gstat[gs_id].gs_memsize > 0` (may be less than requested)

3. Load the guest kernel and binary into memory (`GCTL_LOAD`)

   Required Fields:

   `gs_id` (id of guest returned from `GCTL_RESID`)
   `gs_start` (local buffer containing kernel and param file)
   `gs_ksize` (size of kernel in bytes)
   `gs_psize` (size of parameter file in bytes)

4. Start the guest system (`GCTL_START`)

   Required Fields:

   `gs_id` (id of guest returned from `GCTL_RESID`)

   Optional Fields:

   `gstat[gs_id].gs_halt` (non-zero for halt on guest panic)
   `gs_trace` (non-zero to enable additional kernel tracing)

5. Unlock the guest control structure (`GCTL_UNLOCK`)

*If a guest system has panicked or is stopped and the memory has not yet been released, the id and associated memory may be reused.

To stop a guest system:

1. Lock the guest control structure (`GCTL_LOCK`)
2. Stop the guest kernel (`GCTL_STOP`)

   Required Fields:

   `gs_id` (id of guest returned from `GCTL_RESID`)

3. Unlock the guest control structure (`GCTL_UNLOCK`)

To release guest system memory:

1. Lock the guest control structure (`GCTL_LOCK`)
2. Release the guest memory (`GCTL_RELMEM`)

   Required Fields:

   `gs_id` (id of guest returned from `GCTL_RESID`)

3. Unlock the guest control structure (`GCTL_UNLOCK`)

The guest change (`GCTL_CHANGE`) request can be made at any time to update the following information:

```
gs_trace
gstat[id].gs_halt
gstat[id].gs_owner
gs_cpuhog
gs_schint
gs_winsiz
gs_minres
```

A cooperating guest kernel responds to a `guestctl` freeze request (`GCTL_FREEZE`) by not scheduling user processes. When a thaw is issued (`GCTL_THAW`), normal operation resumes.

1. Lock the guest control structure (`GCTL_LOCK`)
2. Issue the freeze of thaw request (`GCTL_FREEZE` or `GCTL_THAW`)

   Required Fields:

   `gs_id` (id of guest returned from `GCTL_RESID`)

3. Unlock the guest control structure (`GCTL_UNLOCK`)

The system run-level (`GCTL_RUNLVL`) is the only call that can be made from either a host or a guest. It is called by `init`(8) to inform the host of the general system status (single- or multi-user mode).

Required Fields:

`gs_runlvl` (`GSS_SINGLE_USER` or `GSS_MULTI_USER`)

NOTES

The following privileges are required:

Request	**Privilege Required**
GCTL_LOCK	PRIV_ADMIN
GCTL_UNLOCK	PRIV_ADMIN
GCTL_START	PRIV_ADMIN
GCTL_STOP	PRIV_ADMIN
GCTL_STAT	(any user)
GCTL_CHANGE	PRIV_ADMIN
GCTL_RESID	PRIV_ADMIN
GCTL_LOAD	PRIV_ADMIN
GCTL_FREEZE	PRIV_ADMIN
GCTL_THAW	PRIV_ADMIN
GCTL_RELMEM	PRIV_ADMIN
GCTL_RUNLVL	PRIV_RESOURCE
GCTL_RESUME	PRIV_ADMIN
GCTL_GLOAD	PRIV_ADMIN

If the PRIV_SU configuration option is enabled, the super user or a user with the PERMBITS_GUEST permbit is allowed to make any of the guestctl requests.

All users are allowed to make the guest status (GCTL_STAT) request.

CAUTIONS

To avoid the unintentional setting or clearing of fields, it is advisable to obtain a current guest status (GCTL_STATUS) to use as input to the change request.

1. Lock the guest control structure (GCTL_LOCK)
2. Obtain a guest status (GCTL_STAT)
3. Edit fields of interest
4. Issue the change request (GCTL_CHANGE)
5. Unlock the guest control structure (GCTL_UNLOCK)

Although available through the standard UNICOS system call interface, the use of this system call for any request except status (GCTL_STAT) is **not** supported. The system call interface may **change without notice**. See the guest(1) man page for information on managing a guest system.

RETURN VALUES

If `guestctl` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error with a feature specific error. The error, which is returned in the `gctl` status word (`gs_status`), is one of the following:

```
EGS_GSOK        0      /* no error                              */
EGS_SLOTRES     1      /* slot reserved error                   */
EGS_NOROOM1     2      /* insufficient memory                   */
EGS_GBOOT       3      /* gboot struct size difference          */
EGS_GINFOLEN    4      /* ginfo struct size difference          */
EGS_GINFONCPU   5      /* CPU config error                      */
EGS_GINFOCLUS   6      /* cluster config error                  */
EGS_NAMEINUSE   7      /* guest name already in use             */
EGS_NOLOCK      8      /* guestctl lock not held                */
EGS_LOCKED      9      /* guestctl locked already               */
EGS_GINFOGSPEC  10     /* gspec struct size difference          */
EGS_OVERLAP     11     /* memory overlap during load            */
EGS_GADDRESS    12     /* g_address out of range                */
EGS_GCLUSTER    13     /* g_cluster out of range                */
EGS_GENTRY      14     /* g_entry out of range                  */
EGS_GGCOM       15     /* g_gcom out of range                   */
EGS_GERRORP     16     /* g_errorp out of range                 */
EGS_GGPI        17     /* g_gpi out of range                    */
EGS_GGSPEC      18     /* g_gspec out of range                  */
EGS_GNAME       19     /* g_name address range error            */
EGS_GPANICBF    20     /* g_panicbf address range error         */
EGS_GPARK       21     /* g_park address range error            */
EGS_NOID        22     /* no available kernel id                */
EGS_NORELACT    23     /* guest memory release error - active   */
EGS_NORELCHM    24     /* guest memory release error - chm()    */
EGS_MEMREQ      25     /* guest memory request error - chm()    */
EGS_GCPIDLE     26     /* g_cpidle out of range                 */
EGS_GCPUSTATE   27     /* g_cpustate out of range               */
EGS_GCPRMSK     28     /* g_cprmsk out of range                 */
EGS_GTRACEMASK  29     /* g_tracemask out of range              */
EGS_GCLUSREQ    30     /* g_gclusreq out of range               */
EGS_GCSSDREQ    31     /* g_gcssdreq out of range               */
EGS_GCLSYS      32     /* g_clsys out of range                  */
EGS_GPSEMAS     33     /* g_psemas out of range                 */
EGS_GCLMASK     34     /* g_clmask out of range                 */
EGS_GCLSPREQ    35     /* g_gclspreq out of range               */
EGS_BADNAME     36     /* guest name has non-printable chars    */
EGS_HOSTID      37     /* operation not supported on host id    */
EGS_GGBOOT      38     /* g_gboot out of range                  */
```

```
EGS_GINFO       39      /* ginfo out of range                   */
EGS_GGSSDREQ    40      /* gssdreq struct size difference       */
EGS_GCPDOWN     41      /* g_cpdown out of range                */
EGS_GMX         42      /* g_gmx out of range                   */
EGS_GMXL        43      /* gmiop struct size difference         */
EGS_GMXN        44      /* I/O cluster configuration error      */
EGS_GPKTCONF    45      /* g_pktconf out of range               */
EGS_GPKTCONFL   46      /* pktconfig struct size difference     */
EGS_NOROOM2     47      /* insufficient memory                  */
EGS_NOROOM3     48      /* insufficient memory                  */
EGS_NOROOM4     49      /* insufficient memory                  */
EGS_GMEXP       50      /* g_gmexp out of range                 */
EGS_GDESC       51      /* gdesc struct size difference         */
EGS_CONREQ      52      /* invalid number of ttys requested     */
EGS_CONREQ1     53      /* insufficient ttys available          */
EGS_WAITIO      54      /* guest I/O in progress                */
EGS_CPU0DOWN    55      /* CPU 0 is down - must be up for boot  */
EGS_NOPERM      56      /* invalid user permissions             */
EGS_NOSUP       57      /* feature not supported on this H/W    */
EGS_GRPXP       58      /* g_grpxp out of range                 */
EGS_GPDDEREC    59      /* g_pdderec out of range               */
EGS_GPDDERECT   60      /* g_pdderect out of range              */
EGS_GCXTAB      61      /* g_gcxtab out of range                */
EGS_GCXSIZE     62      /* g_gcxsize out of range               */
EGS_ADDMEM      63      /* slot not reserved for add memory req.*/
EGS_NOROOM5     64      /* insufficient memory                  */
EGS_NOCPUS      65      /* no CPUs can be assigned to guest     */
EGS_GBOOTWAIT   66      /* g_bootwait out of range              */
EGS_INVALREQ    67      /* invalid or unsupported guestctl() req*/
EGS_ACTLOAD     68      /* cannot load to active system         */
EGS_NOROOM6     69      /* insufficient memory                  */
```

ERRORS

The following UNICOS system errors may map to one or more specific `guestctl` system call errors (`EGS_xxxxx`).

Error Code	Description
`EACCES`	The request (`gs_req`) is not valid on a guest system.
`EAGAIN`	A required resource is temporarily unavailable. See the `guestctl` error for more information.

EBUSY	A release of guest memory cannot yet be performed due to the possibility of outstanding I/O from the previously active guest. See the guestctl error for more information.
EDEADLK	The guestctl is currently held by another process.
EFAULT	A copy of data to or from the user area failed. See the guestctl error for more information.
EINVAL	The request number (gs_req) may not be valid or the specified guest id (gs_id) is greater than MAXGUESTS. See the guestctl error for more information.
ENOMEM	Mainframe memory is not currently available to satisfy the GCTL_RESID request. See the guestctl error for more information.
EPERM	The caller does not have the PRIV_ADMIN privilege. The caller does not have the PERMBITS_GUEST permbit.

FILES

/etc/udb	Contains a list of valid users and lists their privileges and permissions

SEE ALSO

guest(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

init(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`ialloc` – Allocates storage for a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/file.h>
#include <unistd.h>
```

`long ialloc (int` *fildes*`, long` *nb*`, int` *flag*`, int` *part*`, long *`*avl*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `ialloc` system call provides the means to preallocate storage for a file with certain user-specified constraints. These include mandatory contiguous storage and the partition of the file system in which to begin the search. The new space is allocated at the end of the file.

The `ialloc` system call accepts the following arguments:

fildes Specifies a file descriptor. It is obtained from a `creat`(2), `dup`(2), `fcntl`(2), or `open`(2) system call.

nb Specifies the number of bytes to allocate.

flag Controls allocation. The following are valid values for *flag*:

- `IA_CONT` Allocates contiguous storage only; if unavailable, returns error.
- `IA_PART` Allocates partition specified by *part*. If *cbits* was specified at file creation time (see `open`(2)), allocates space on the partitions specified by that argument.
- `IA_BEST` If all the blocks cannot be allocated as specified, allocates as much as possible.
- `IA_RAVL` If allocation is successful, stores number of bytes actually allocated at *avl*. If `IA_CONT` is set and allocation is unsuccessful, stores maximum number of bytes that could have been allocated at *avl*.

part Specifies the partition in which allocation is attempted.

avl Points to where the number of bytes actually allocated is stored, if `IA_RAVL` is specified.

NOTES

The process must be granted write permission to the file via the security label. That is, the active security label of the process must be equal the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_WRITE	The process is granted write permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted write permission to the file via the security label.

RETURN VALUES

If ialloc completes successfully, *nb* is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The ialloc system call fails if one of the following error conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not a valid file descriptor open for writing, or *fildes* is not a regular file in the native file system (NC1FS or SFS).
EBADF	The security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
EFAULT	*avl* points outside the program address space.
EFBIG	An attempt was made to allocate a file that exceeds the file size limit or the maximum file size of the process. See ulimit(2).
EINVAL	*flag* value not defined.
ENOSPC	During the allocation, no free space was found in the file system.
EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.

FORTRAN EXTENSIONS

The ialloc system call can be called from Fortran as a function:

```
INTEGER fildes, nb, flag, part, avl, IALLOC, I
I = IALLOC (fildes, nb, flag, part, avl)
```

Alternatively, ialloc can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes, nb, flag, part, avl
CALL IALLOC (fildes, nb, flag, part, avl)
```

The Fortran program must not specify both the subroutine call and the function reference to ialloc from the same procedure.

EXAMPLES

The following examples illustrate different uses of the ialloc system call.

Example 1: This ialloc request attempts to preallocate 10 data blocks (4096 bytes each) to the newly created file test_file. If the requested amount of contiguous space is unavailable, the request fails.

```
#define BLK_SZ  4096

int fd;

fd = open("test_file", O_WRONLY | O_CREAT, 0644);

if (ialloc(fd, 10*BLK_SZ, IA_CONT, 0, (long *) 0) == -1) {
     perror("ialloc failed to allocate 10 blocks contiguously");
     exit(1);
}
```

Example 2: This ialloc request attempts to preallocate 100,000 bytes to the newly created file datafile in the third partition of the file system in which the file resides. If insufficient space exists to allocate the file in this partition, the allocation is attempted in other partitions of the file system. (File system partitions are numbered 0 –*n*.) A contiguous allocation is not required since the IA_CONT flag is not specified.

```
int fd;

fd = open("datafile", O_WRONLY | O_CREAT, 0600);

if (ialloc(fd, 100000, IA_PART, 2, (long *) 0) == -1) {
     perror("ialloc failed for file datafile");
     exit(1);
}
```

FILES

/usr/include/unistd.h	Contains C prototype for the ialloc system call

SEE ALSO

creat(2), dup(2), fcntl(2), open(2), ulimit(2)

NAME

ioctl – Controls device

SYNOPSIS

```
#include <sys/ioctl.h>
int ioctl (int fildes, int request, int arg);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The ioctl system call performs a variety of functions on character special files (devices). It accepts the following arguments:

fildes Specifies a file descriptor of a special file. It is obtained from an accept(2), dup(2), fcntl(2), open(2), socket(2), or socketpair(2) system call

request Specifies a command to be issued to the device driver.

arg Specifies an argument to the *request* command passed to the device driver.

The descriptions of various devices in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, discuss how ioctl applies to them.

NOTES

Only a process with appropriate privilege can control a restricted device.

To retrieve information about certain devices, the active security label of the process must be greater than or equal to the security label of the device file.

To set information about certain devices, the active security label of the process must equal the security label of the device file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_IO	The process is allowed to control a restricted device.
PRIV_MAC_READ	The process is allowed to retrieve information about certain devices regardless of the security label of the device file.
PRIV_MAC_WRITE	The process is allowed to set information about certain devices regardless of the security label of the device file.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override the security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to control a restricted device.

RETURN VALUES

If `ioctl` completes successfully, it returns an integer value that depends on the device control function; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `ioctl` system call fails if one of the following error conditions occurs:

Error Code	Description
`EBADF`	The *fildes* argument is not a valid open file descriptor.
`EBADF`	The process does not meet security label requirements and does not have appropriate privilege.
`EINVAL`	The *request* or *arg* argument is not valid.
`ENOTTY`	The *fildes* argument is not associated with a character special device.
`EPERM`	The process does not have appropriate privilege to control a restricted device.

For information about specific devices, see the appropriate entry in section 4 in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.

FORTRAN EXTENSIONS

The `ioctl` system call can be called from Fortran as a function:

```
INTEGER fildes, request, arg, IOCTL, I
I = IOCTL (fildes, request, arg)
```

Alternatively, `ioctl` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes, request, arg
CALL IOCTL (fildes, request, arg)
```

The Fortran program must not specify both the subroutine call and the function reference to `ioctl` from the same procedure.

EXAMPLES

The following examples illustrate how to use the `ioctl` system call to control two (terminals and CPUs) of the many character devices that `ioctl` can control.

Example 1: Terminals typically operate in line mode, meaning that a process reading data from the terminal (like a shell program) does not receive any data until the user enters a line terminator (usually a CR character).

This program disables that characteristic such that after the user enters only 2 characters, the reading process receives the 2 characters.

For additional information on this topic and other capabilities, refer to `termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.

```
#include <termio.h>

main()
{
     struct termio termo, termn;

     if (ioctl(1, TCGETA, &termo) == -1) {
          perror("ioctl (TCGETA) failed getting terminal parameters");
          exit(1);
     }

     termn = termo;             /* copy old terminal parameters to new */
     termn.c_lflag &= ˜ICANON; /* disable canonical terminal mode */
     termn.c_cc[VMIN] = 2;     /* characters sent after 2 typed */
     termn.c_cc[VTIME] = 100;  /* specify 10 second delay between keystrokes */

     if (ioctl(1, TCSETA, &termn) == -1) {
          perror("ioctl (TCSETA) failed setting new terminal parameters");
          exit(1);
     }

     /* After the ioctl request changed the terminal parameters,
        the user interface changed.  The terminal is no longer in
        line mode (canonical).  After 2 keystrokes with or without
        a CR character, the 2 characters are delivered to the reading
        process. */

     /* Before the program terminates, the terminal’s parameters are restored
        to their original state. */

     if (ioctl(1, TCSETA, &termo) == -1) {
          perror("ioctl (TCSETA) failed resetting terminal parameters");
          exit(1);
     }
}
```

Example 2: An `ioctl` system call can control a CPU in a variety of ways. This `ioctl` request causes the CPU to interrupt the currently running program with a `SIGALRM` signal at regularly scheduled intervals (measured in milliseconds). The program catches each signal at the defined intervals and continues.

For additional information on this topic and other capabilities, refer to `cpu`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.

```
#include <fcntl.h>
#include <sys/cpu.h>
#include <signal.h>
#include <time.h>

long before, after;

main()
{
     int fd;
     struct cpudev cpudev;
     void catch(int signo);

     (void) signal(SIGALRM, catch);

     fd = open("/dev/cpu/any", O_RDONLY);

     cpudev.word = 10000;  /* set interval to 10 seconds (10,000 mill) */
     if(ioctl(fd, CPU_SETTMR, &cpudev) == -1) {
          perror("ioctl failed");
          exit(1);
     }

     before = rtclock();   /* read the real time clock */

     for(;;) {             /* loop indefinitely waiting for SIGALRMs
                              every 10 sec */
     }                     /* kill with <ctrl><C> */
}

void catch(int signo)
{
     float time;

     (void) signal(signo, catch);

     after = rtclock();                                  /* read the real
                                                            time clock */
     time = (float) (after - before) / (float) CLK_TCK;  /* compute seconds */
     printf("Caught signal #%d after %f seconds\n", signo, time);
}
```

FILES

`/usr/include/sys/ioctl.h` Contains C prototype for the `ioctl` system call

SEE ALSO

`accept`(2), `dup`(2), `fcntl`(2), `open`(2), `socket`(2), `socketpair`(2)

`cpu`(4), `termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

jacct – Enables or disables job accounting

SYNOPSIS

```
#include <unistd.h>
int jacct (char *path);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The jacct system call enables or disables job accounting for the calling job (see setjob(2)). If job accounting is enabled, an accounting record is written on a job accounting file for each of the job's processes that terminates. If daemon accounting is enabled, daemon accounting records are also written to this file. An exit(2) call or a signal can cause termination. Any process member of a job may use the jacct call.

The jacct system call accepts the following argument:

path Points to a path name that contains the job accounting file. acct(5) and /usr/include/acct/dacct.h describe the types of records found in this file.

Job accounting is enabled if *path* is nonzero and no errors occur during the system call. It is disabled if *path* is 0 and no errors occur during the system call.

If job accounting is already enabled and *path* differs from the job accounting file currently in use, the job accounting file will be switched to *path* without the loss of any accounting information.

NOTES

To be granted write permission to the file, the active security label of the process must equal the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_DAC_OVERRIDE	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label.
PRIV_MAC_READ	The process is granted read permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted read permission to the file via the security label.

RETURN VALUES

If jacct completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The jacct system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the *path* prefix denies write permission.
EACCES	The file specified by *path* is not an ordinary file.
EACCES	The write permission is denied for the specified job accounting file.
EFAULT	The *path* argument points to an illegal address.
EINVAL	The calling process is not a member of a job.
EISDIR	The specified file is a directory.
ENOENT	One or more components of the job accounting file path name do not exist.
ENOTDIR	A component of the *path* prefix is not a directory.
EROFS	The specified file resides on a read-only file system.

FILES

/usr/include/unistd.h	Contains C prototype for the jacct system call

SEE ALSO

acct(2), dacct(2), exit(2), setjob(2), signal(2)

acct(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

join, fjoin – Joins files

SYNOPSIS

```
#include <unistd.h>
int join (char *path1, char *path2);
int fjoin (int fildes1, int fildes2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The join system call concatenates the data blocks of one file to another. The join system call accepts the following arguments:

path1 Points to the file to which you want to add data blocks.

path2 Points to the file from which you want to take data blocks away.

Data blocks are not copied from the file referenced by *path2* to that referenced by *path1*; rather the address descriptors in the inode for the second file are appended to the address descriptors in the inode for the first.

Any allocated, but unused data blocks at the end of the file identified by *path1* are deallocated prior to the addition of data blocks from address descriptors in the inode for the file identified by *path2*. The length of the file identified by *path2* is truncated to 0 by the system call.

The fjoin system call performs the same operation as join with the specified files. The fjoin system call accepts the following arguments:

fildes1 Specifies the file descriptor of the file to which you want to add data blocks.

fildes2 Specifies the file descriptor of the file from which you want to take data blocks away.

RETURN VALUES

If join or fjoin completes successfully, 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The join system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the *path1* or *path2* prefix denies search permission.
EACCES	Write permission is denied for the file specified by *path1* or *path2*.

`EAGAIN`	Mandatory file and record locking is set, and there are outstanding record locks on one of the files (see `chmod`(2)).
`EFAULT`	The *path1* or *path2* argument points outside the allocated process address space.
`EINVAL`	*path1* and *path2* identify the same file.
`EINVAL`	Files reside on different file systems.
`EINVAL`	*path1* or *path2* does not identify a regular file.
`EINVAL`	The length of file identified by *path1* is not an even multiple of 4096 bytes, not an even multiple of the physical I/O unit size of the device on which the file system resides, nor an even multiple of the file system allocation unit size of the partition on which the file resides.
`ENOENT`	The file identified by *path1* or *path2* does not exist.
`ENOTDIR`	A component of the *path1* or *path2* prefix is not a directory.

The `fjoin` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Write permission is denied for the file specified by *fildes1* or *fildes2*.
`EAGAIN`	Mandatory file and record locking is set, and there are outstanding record locks on one of the files (see `chmod`(2)).
`EBADF`	The calling process does not have MAC read access to the file to which the file descriptor refers.
`EINVAL`	*fildes1* and *fildes2* identify the same file.
`EINVAL`	Files reside on different file systems.
`EINVAL`	*fildes1* or *fildes2* does not identify a regular file.
`EINVAL`	The length of file identified by *fildes1* is not an even multiple of 4096 bytes, not an even multiple of the physical I/O unit size of the device on which the file system resides, or an even multiple of the file system allocation unit size of the partition on which the file resides.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `join` and `fjoin` system calls

SEE ALSO

`chmod`(2)

`mkfs`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`kill`, `killm`, `_lwp_kill`, `_lwp_killm` – Sends a signal to a process or a group of processes

SYNOPSIS

All Cray Research systems:

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int sig);

#include <sys/category.h>
#include <signal.h>

int killm (int category, int id, int sig);
```

Cray PVP systems:

```
#include <sys/types.h>
#include <signal.h>

int _lwp_kill (pid_t pid, int sig);

#include <sys/category.h>
#include <signal.h>

int _lwp_killm (int category, int id, int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `kill`)

DESCRIPTION

The `kill` system call sends a signal to a process or a group of processes. The `kill` and `_lwp_kill` system calls accept the following arguments:

pid Specifies the process or group of processes to which the signal is to be sent.

sig Specifies the signal that is to be sent. Specify either 0 or one of the values for the *sig* argument for the `signal`(2) system call. If *sig* is 0 (the null signal), error checking is performed, but no signal is sent. You can use this to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the sending process has appropriate privilege. The caller can send a `SIGCONT` signal to any process within its session, regardless of the process owner.

The processes with a process ID of 0 and a process ID of 1 are special processes (see `intro`(2)), and they are referred to as `proc0` and `proc1`, respectively, in the following conditions:

- If *pid* is greater than 0, *sig* is sent to the process that has a process ID equal to *pid*; *pid* can equal 1.
- If *pid* equals 0, *sig* is sent to all processes, excluding `proc0` and `proc1`, whose process group ID is equal to the process group ID of the sender.
- If *pid* equals −1 and the sending process has appropriate privilege, *sig* is sent to all processes excluding `proc0` and `proc1.`
- If *pid* equals −1 and the sending process does not have appropriate privilege, *sig* is sent to all processes, excluding `proc0` and `proc1`, whose real user ID is equal to the effective user ID of the sender.
- If *pid* is negative but not −1, *sig* is sent to all processes whose process group ID is equal to the absolute value of *pid*.

The `killm` system call sends a signal to a process or a group of processes. The `killm` and `_lwp_killm` system calls accept the following arguments:

category Specifies `C_PROC`, `C_PGRP`, `C_ALL`, `C_UID`, or `C_JOB`. A category of `C_ALL` is available only when the sending process has the appropriate privilege.

id Specifies the *pid*, *pgrp*, *jid*, or *uid* corresponding to the *category*. An *id* of 0 means all processes in the current *category*.

sig Identifies the signal to be sent. See `signal`(2) for *sig* values.

Currently the `_lwp_kill` and `_lwp_killm` interfaces are synonyms for `kill` and `killm`, respectively. In a future release, this will change (see the NOTES section).

If the target of a `kill` or `killm` system call is an MPP application, each processing element (PE) in the application receives the signal.

NOTES

For multitasked applications, the `kill` and `killm` system calls treat the entire multitasking group as the target of a signal. More specifically, the system decides which member of the multitasking group receives the signal and the *pid* argument is not considered significant in this choice (although it may introduce some bias in the selection).

In contrast, the `_lwp_kill` and `_lwp_killm` system calls do consider the *pid* argument as an explicit specifier of the receiver of a signal. However, use of these calls is discouraged since they may disappear in future releases of the UNICOS operating system.

Signals are not allowed to cross security label boundaries unless the sending process has privilege to override the system mandatory access control (MAC) policy. If an unprivileged process attempts to send a signal to another process that has a different security label, an `ESRCH` error status is returned.

The active security label of the process must equal the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_KILL`	The process is considered the owner of all affected processes.
`PRIV_MAC_WRITE`	The active security label of the process is considered to equal the security label of all affected processes.

If the `PRIV_SU` configuration option is enabled, the super user is considered the owner of all affected processes. If the `PRIV_SU` configuration option is enabled, the super user overrides all security label restrictions.

RETURN VALUES

If `kill`, `killm`, `_lwp_killm`, or `_lwp_killm` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `kill` or `killm` system call fails and no signal is sent if one of the following error conditions occurs:

Error Code	Description
`EINVAL`	The *sig* argument is not a valid signal number.
`EPERM`	The *pid* argument is 1 (`proc1`), and *sig* is either `SIGKILL` or `SIGSTOP`.
`EPERM`	The process does not have appropriate privilege, and its real or effective user ID does not match the real or effective user ID of the receiving process.
`ESRCH`	The active security label of the process does not equal those of a receiving process, and the process does not have appropriate privilege.
`ESRCH`	No process can be found corresponding to that specified by *pid*.

FORTRAN EXTENSIONS

The `kill` system call can be called from Fortran as a function:

```
INTEGER pid, sig, KILL, I
I = KILL (pid, sig)
```

Alternatively, `kill` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable:

```
INTEGER pid, sig
CALL KILL (pid, sig)
```

The Fortran program must not specify both the subroutine call and the function reference to `kill` from the same procedure.

EXAMPLES

The following examples illustrate the use of the `kill` system call and `killm`, the Cray Research extension. Each example entails sending a `SIGUSR1` signal to the parent of the calling process:

Example 1: The `kill` request sends a `SIGUSR1` signal to the parent process:

```
int ppid;

ppid = getppid();
if (kill(ppid, SIGUSR1) == -1) {
     perror("kill failed sending SIGUSR1 to parent");
     exit(1);
}
```

Example 2: The `killm` request sends a `SIGUSR1` signal to the parent process:

```
int ppid;

ppid = getppid();
if (killm(C_PROC, ppid, SIGUSR1) == -1) {
     perror("killm failed sending SIGUSR1 to parent");
     exit(1);
}
```

SEE ALSO

`getpid`(2), `intro`(2), `setpgrp`(2), `signal`(2)

`kill`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`limit` – Sets resource limits

SYNOPSIS

```
#include <sys/category.h>
#include <sys/resource.h>
```

`long limit (int` *category*`, int` *id*`, int` *resource*`, long` *newlimit*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `limit` system call establishes limits on resource usage or returns information on resource limits for a process or job. It accepts the following arguments:

category Selects the resource category. The following are valid resource categories:

- `C_JOB` Job limit.
- `C_JOBPROCS` Process limit of all processes associated with job *jid*.
- `C_PROC` Process limit.

id Specifies the *pid* or *jid* corresponding to the *category*. A *pid* of 0 means the current process, and a *jid* of 0 means the current job.

resource Selects the resource. The following are valid resources:

- `L_CORE` Maximum core file limit (clicks). A *newlimit* value less than the process size will result in a truncated core file consisting of the user common structure and the user area. A value of `NO_CORE_FILES` will disable the creation of core files altogether. This limit is supported only for the `C_PROC` category.
- `L_CPROC` Maximum number of processes that can exist concurrently within a job. This limit is supported only for the `C_JOB` category.
- `L_CPU` Maximum CPU time per category (clocks). If a process exceeds the process limit, `SIGCPULIM` is sent. The parent shell recognizes the death of the process and sends an error message to standard error. If the job limit is exceeded, `SIGCPULIM` is sent to all processes in the job, which includes the parent shell. Processes may register to catch this signal and continue, but `SIGKILL` is sent a few seconds later. (See the CAUTIONS section.)

`L_FD`	Maximum number of open files that children of this process will have when created. This limit is supported only for the `C_PROC` category. If the new limit is less than the value of `OPEN_MAX` (64), the limit will be set to `OPEN_MAX` and no error will be returned. The specified limit must be less than that set by the `L_FDM` resource or the system-imposed open file maximum value of `K_OPEN_MAX`.
`L_FDM`	Maximum limit on the `L_FD` resource setting (the minimum limit is set by `OPEN_MAX`). Changing the `L_FDM` resource does not affect the open file maximum of any processes. Rather, it affects the open file maximum of any future child processes by limiting the maximum `L_FD` resource specification. This limit is supported only for the `C_PROC` category. If the new limit is less than the specified process' current open file maximum, `limit` will fail with an `EINVAL` error status. If the new limit is greater than the system imposed `K_OPEN_MAX` open file limit, `limit` will set the limit to `K_OPEN_MAX`.
`L_FSBLK`	Maximum number of file system blocks (clicks) that can be used per category. If a process tries to exceed established process or job limits, an `EDISKLIM` error is returned.
`L_MEM`	Maximum memory size per category (clicks). If a process tries to exceed established process or job limits, the `brk`(2) or `sbrk`(2) system call fails and returns the `ENOMEM` error.
`L_MPPB`	(Deferred implementation) Maximum number of Cray MPP synchronization barriers. This limit is supported only for the `C_JOB` category.
`L_MPPE`	Maximum number of Cray MPP processing elements (PEs). If this limit is set to 0, the Cray MPP systems cannot be used by the job. This limit is supported only for the `C_JOB` category.
`L_MPPT`	Maximum number of wall clock seconds that the job can have the Cray MPP systems assigned to it. If this limit is set to 0, the Cray MPP systems cannot be used by the job. This limit is supported for all categories.
`L_SDS`	Maximum number of secondary data blocks per category. It is enforced by the system on an `ssbreak`(2) call. Because the minimum allocation unit for secondary data segments (SDS) may be greater than 1 block, the limit set may be exceeded by a fraction of the minimum allocation unit.
`L_SOCKBF`	Maximum total socket buffer (sockbuf) space per session. The per session sockbuf space is the sum of the sockbuf space reserved by all of the sockets used by the session. The limit is in clicks (4096 bytes per click). This limit is enforced on the `accept`(2), `setsockopt`(2), and `socket`(2) calls.
`L_TAPE`	Maximum number of tape devices from tape group 0 for the job. This is a synonym for `L_TAPE0` and will be removed in a future release. It is enforced by the tape daemon. This limit is supported only for the `C_JOB` category.

`L_TAPE`*n*	Maximum number of tape devices from tape group *n* (which can be 0 through 7) for the job. It is enforced by the tape daemon. This limit is supported only for the `C_JOB` category.

newlimit Specifies a new limit. *newlimit* is one of the following:

–1	Limit unchanged; current limit value is returned.
≥0	New limit value. For all limits except tape group limits, `L_SDS`, `L_MPPB`, `L_MPPE`, and `L_MPPT`, a value of 0 means no limit.
–2	For `L_CORE` only. This special value disables the creation of core files.

Any process can make a limit more restrictive, but only a process with appropriate privilege can make a limit less restrictive. Limits are inherited by child processes.

NOTES

The following mandatory access control (MAC) read and MAC write checks are performed based on the *category* parameter:

Parameter	**Description of check**
`C_PROC`	Against the specified process
`C_JOBPROCS`	Against each process in the job
`C_JOB`	Against the job leader

That is, the active security label of the calling process must equal the security label of each process where access is being verified.

To set process resource information, the active security label of the calling process must equal the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_READ`	The calling process is allowed to override the restriction that its active security label must be greater than or equal to the security label of every affected process.
`PRIV_MAC_WRITE`	The calling process is allowed to set resource information regardless of the security label of the target process.
`PRIV_POWNER`	The process is considered the owner of every affected process.
`PRIV_RESOURCE`	The calling process is allowed to increase the value of a limit.

If the `PRIV_SU` configuration option is enabled, the super user is considered the owner of every affected process and is allowed to increase the value of a limit. If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security label restrictions.

CAUTIONS

The CPU time limit does not apply when running as root.

RETURN VALUES

If `limit` completes successfully, the previous value of `limit` is returned for categories `C_PROC` and `C_JOB`, and 0 is returned for `C_JOBPROCS`; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `limit` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EBADF`	The process does not meet security label requirements and does not have appropriate privilege.
`EINVAL`	One of the arguments contains an invalid value.
`EPERM`	The user ID of the requesting process is not that of a super user, and its real or effective user ID does not match the real or effective user ID of the affected processes.
`EPERM`	An attempt was made to increase the value of a limit, and the user ID of the requesting process is not that of a super user.
`EPERM`	An attempt was made to change a limit on a recovered job, recovered process, or a system process; this is not allowed.
`ESRCH`	No process can be found that matches the *category* and *id* requests.

FORTRAN EXTENSIONS

The `limit` system call can be called from Fortran as a function:

```
INTEGER category, id, resource, newlimit, LIMIT, I
I = LIMIT (category, id, resource, newlimit)
```

Alternatively, `limit` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER category, id, resource, newlimit
CALL LIMIT (category, id, resource, newlimit)
```

The Fortran program must not specify both the subroutine call and the function reference to `limit` from the same procedure.

EXAMPLES

The following example shows how to use the `limit` system call to return the current maximum CPU time and memory size limits for the calling process. Neither `limit` request changes the CPU time or memory size limit because of the argument value –1 specified.

```
#include <sys/category.h>
#include <sys/resource.h>

main()
{
     long time, mem;

     time = limit(C_PROC, 0, L_CPU, -1);
     printf("The current CPU time limit is %ld clock ticks or %ld sec\n",
             time, time/CLK_TCK);        /* CLK_TCK defined in time.h> */

     mem = limit(C_PROC, 0, L_MEM, -1);
     printf("The current memory limit is %ld blocks, %ld words, or %.1f Mw\n",
             mem, mem * 512, (mem * 512)/1048576.);
}
```

SEE ALSO

`accept`(2), `brk`(2), `setsockopt`(2), `signal`(2), `socket`(2), `ssbreak`(2), `ulimit`(2)

`limit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`limits` – Returns or sets limits structure for fair-share scheduler

SYNOPSIS

```
#include <sys/types.h>
#include <sys/lnode.h>
#include <sys/param.h>
#include <sys/iosw.h>
#include <sys/signal.h>
#include <sys/dir.h>
#include <sys/perm.h>
#include <sys/retlim.h>
#include <sys/share.h>
```

`int limits (struct lnode *`*address*`, int` *function*`);`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `limits` system call manipulates a kernel limits structure according to the value of *function*. `limits` accepts the following arguments:

address Points to an lnode or an array of lnodes except where indicated.

function Specifies a function. *function* may be one of the following:

`L_MYLIM` Gets user's own limits structure. Returns the number of processes currently attached to the node.

`L_OTHLIM` Gets limits associated with *uid* in `lnode`. The lnode to which *address* points must contain the correct user ID. Returns the number of processes currently attached to the node.

`L_ALLLIM` Returns the number of active lnodes, along with all active user limits structures.

`L_SETLIM` Connects to a new limits structure. Initializes a new limits structure with the passed lnode, and attaches the calling process to it. All children of that process inherit the new structure. Only a process with appropriate privilege can specify this function.

`L_NEWLIM` Same as `L_SETLIM`, but attaches parent processes rather than calling processes. Only a process with appropriate privilege can specify this function.

`L_DEADLIM` Waits for dead limits belonging to a child process. The *address* should point to a `retlim` structure, which is defined in the `sys/retlim.h` file. This function performs a `wait`(2) system call, then returns a structure containing both the limits and process zombie structures. The value returned is the number of processes still attached to the lnode.

`L_CHNGLIM` Changes `limits` fields in existing limits. The lnode to which *address* points must contain the correct user ID. Only a process with appropriate privilege can specify this function.

NOTE: This function updates the CPU quota-used field. To synchronize the information in active lnodes with the user database (UDB), execute the `shrsync`(8) command with the `-q` option.

`L_DEADGROUP` Picks up a dead limits structure. This function searches for a dead limits structure, removes it from the list of active limits, and returns `lnode`. Only a process with appropriate privilege can specify this function.

`L_MSGSON` Enables system messages to a user.

`L_MSGSOFF` Disables system messages to a user; default is enabled.

`L_MSGSSTAT` Returns the status of system messages; if system messages are enabled, a nonzero number is returned.

`L_MYKN` Gets a user's own `kern_lnode` structure. The *address* should point to a `kern_lnode` structure, which is defined in the `sys/lnode.h` file. Returns the number of processes currently attached to the node.

`L_OTHKN` Gets the structure associated with *uid*. The *address* should point to a `kern_lnode` structure, which is defined in the `sys/lnode.h` file. The `kern_lnode` to which *address* points returns the number of processes currently attached to the node.

`L_ALLKN` Returns the number of active lnodes, along with all active kernel structures. The *address* should point to a `kern_lnode` structure, which is defined in the `sys/lnode.h` file.

`L_SETIDLE` Sets `limits` fields for idle lnode, which is initialized at system boot time; otherwise, it acts as `L_SETLIM` does. Only a process with appropriate privilege can specify this function.

Any other *function* is illegal and returns an error of `EINVAL`. Unless otherwise specified, the call returns the number of limits structures returned.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_RESOURCE	The process is allowed to specify the L_SETLIM, L_CHNGLIM, L_DEADGROUP, and L_SETIDLE functions.

If the PRIV_SU configuratino option is enabled, the super user or a process with the PERMBITS_RESLIM permbit is allowed to specify the L_SETLIM, L_CHNGLIM, L_DEADGROUP, and L_SETIDLE functions.

The L_GETCOSTS and L_SETCOSTS functions have been removed in the UNICOS 9.0 release. Their functionality has been replaced by the GET_COSTS and SET_COSTS actions, respectively, of the policy(2) system call.

RETURN VALUES

If limits completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The limits system call fails if one of the following error conditions occurs:

Error Code	Description
EINVAL	An illegal argument was passed to the system call.
EPERM	For functions L_SETLIM, L_CHNGLIM, L_DEADGROUP, and L_SETIDLE, this error indicates that the process does not have appropriate privilege.
EPROCLIM	The user already has the maximum number of processes active (valid for function L_SETLIM).
ESRCH	For functions L_DEADGROUP, L_OTHKN, L_OTHLIM, and L_CHNGLIM, this error indicates that the desired limits structure does not exist. For function L_SETLIM, this error indicates that this lnode's group has not been set up.
ETOOMANYU	No space is left in the kernel limits table (valid for function L_SETLIM).

SEE ALSO

policy(2), wait(2)

share(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

shrsync(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

`link` – Creates a link to a file

SYNOPSIS

```
#include <unistd.h>
int link (const char *path1, const char *path2);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `link` system call creates a new link (directory entry) for the existing file. Its accepts the following arguments:

path1 Points to a path name identifying an existing file.

path2 Points to a path name identifying the directory entry to be created.

NOTES

The process must be granted search permission to every component of each path prefix via the permission bits and access control list. The process must be granted search permission to every component of each path prefix via the security label.

The process must be granted write permission to the parent directory of *path2* via the permission bits and access control list. The process must be granted write permission to the parent directory of *path2* via the security label.

If *path1* is a directory, the process must have appropriate privilege to create a link.

If `FSETID_RESTRICT` is enabled, only processes with appropriate privileges can create a link to set-user-ID or set-group-ID files.

A process with the effective privileges shown are granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of each path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the parent directory of *path2* via the permission bits and access control list.

`PRIV_FSETID`	If `FSETID_RESTRICT` is enabled, the process can create a link to the set-user-ID or set-group-ID file.
`PRIV_LINK_DIR`	The process can create a link to a directory.
`PRIV_MAC_READ`	The process is granted search permission to every component of each path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the parent directory of *path2* via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of each path prefix and is granted write permission to the parent directory of *path2*. The super user can create a link to a directory. The super user or a process with the `suidgid` permission can override the restriction enabled by the `FSETID_RESTRICT` system configuration option.

RETURN VALUES

If `link` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `link` system call fails and no link is created if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of either path prefix denies search permission.
`EACCES`	The requested link requires writing in a directory with a mode that denies write permission.
`EEXIST`	The link specified by *path2* exists.
`EFAULT`	The *path* argument points outside the allocated address space of the process.
`EMANDV`	The security label of the file does not allow linking.
`EMANDV`	If the `FSETID_RESTRICT` and `PRIV_SU` configuration options are enabled, the process does not have appropriate privileges to link to a set-user-ID or set-group-ID file.
`EMLINK`	The maximum number of links to a file (`LINK_MAX`) would be exceeded.
`ENOENT`	A component of either path prefix does not exist.
`ENOENT`	The file specified by *path1* does not exist.
`ENOENT`	The *path2* argument points to a null path name.
`ENOTDIR`	A component of either path prefix is not a directory.
`EPERM`	The file specified by *path1* is a directory, and the effective user ID is not that of a super user.

EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.
EROFS	The requested link requires writing in a directory on a read-only file system.
EXDEV	The link specified by *path2* and the file named by *path1* are on different logical devices (file systems).

FORTRAN EXTENSIONS

The `link` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER path1*M, path2*N
INTEGER LINK, I
I = LINK (path1, path2)
```

Alternatively, `link` can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER path1*M, path2*N
CALL LINK (path1, path2)
```

The Fortran program must not specify both the subroutine call and the function reference to `link` from the same procedure. *path1* and *path2* may also be integer variables. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFLINK`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example shows how to use the `link` system call to create a link to another user's file. The following `link` request creates a link (called `joe_file`) to user `joe`'s file, `datafile`.

The request causes a new directory entry named `joe_file` to be created and the inode for `joe`'s `datafile` to reflect this additional link.

If user `joe` later removes (by using `rm`(1) or `unlink`(2)) `datafile`, the file is not actually removed from the file system since another user now has a link to it. The file cannot be removed until the last link to the file is removed.

```
if (link("../joe/datafile", "joe_file") == -1) {
     perror("link failed creating new link to joe's datafile");
     exit(1);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `link` system call

SEE ALSO

`unlink`(2)

`rm`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`PXFLINK`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

`link`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`listen` – Listens for connections on a socket

SYNOPSIS

`int listen (int` *s*`, int` *backlog*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

To accept connections, a socket must first be created with `socket`(2). A backlog for incoming connections is then specified with `listen`, and the connections are accepted with `accept`(2). The `listen` call applies only to sockets of type `SOCK_STREAM`.

Connection requests for an address go into the connection queue for the socket bound to the specified address. The `accept`(2) system call removes connection requests from the queue.

The `listen` system call accepts the following arguments:

s Specifies the descriptor for a socket.

backlog Defines the maximum length to which the queue of pending connections can grow. If a connection request arrives with the queue full, the client might receive an error with an indication of `ECONNREFUSED`, or if the underlying protocol supports retransmission, the request might be ignored so that retries can succeed.

The kernel has a limit for the maximum length of the queue of pending connections. If the *backlog* parameter exceeds that limit, the kernel limit is used instead. However, the kernel also allows some number of connections beyond the queue limit to be accepted, to allow for transient connections that never get established fully.

Note: The maximum limit is defined by `SOMAXCONN` in the `sys/socket.h` file. Currently, the maximum for queued pending connections is (*backlog* * 3)/2 +1.

NOTES

If the `SOCKET_MAC` option is enabled, the active security label of the process must be greater than or equal to the security label of the socket. Note that `SOCKET_MAC` is part of TCP/IP configurable feature variables list in `uts/cf/Nmakefile`. For more information, see the `connect`(2) man page.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_READ`	The process is allowed to override the security label restrictions when the `SOCKET_MAC` option is enabled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions when the SOCKET_MAC option is enabled.

RETURN VALUES

If listen completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The listen system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	If the SOCKET_MAC option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EBADF	The *s* descriptor is invalid.
ENOTSOCK	The *s* descriptor is not a socket.
EOPNOTSUPP	The socket is not of a type that supports the operation listen (for example, the socket is of type SOCK_DGRAM).

BUGS

Currently, the backlog is limited (silently) to five pending connection requests by SOMAXCONN (SOMAXCONN = 5 is defined in the sys/socket.h include file). When the number entered in the *backlog* argument is higher than 5, no error message is issued.

EXAMPLES

This server program shows how to use the listen system call in context with other TCP/IP calls. (Some system calls in this example are not supported on Cray MPP systems.) The program simply creates a TCP/IP socket, waits for a client process from some host to attempt a connection, accepts the connection, and forks a child process to provide the service to the client.

The original (parent) server loops back to look for additional connection attempts while the temporary (child) server reads a string of data sent by the client process.

```
/*  Server side of client-server socket example. For client side,
    see socket(2).
    Syntax: server portnumber & */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

main(int argc, char *argv[])
{
     int s, ns;
     struct sockaddr_in src;  /* source socket address */
     int len=sizeof(src);
     char buf[256];

     /* create port */
     src.sin_family = AF_INET;
     src.sin_port = atoi(argv[1]);
     src.sin_addr.s_addr = 0;

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("server, unable to open socket");
          exit(1);
     }

     while (bind(s, (struct sockaddr *) &src, sizeof(src)) < 0) {
          printf("Server waiting on bind...\n");
          sleep(1);
     }

     listen(s, 5);

     while (1) {
          ns = accept(s, (struct sockaddr *) &src, &len);
          if (ns < 0) {
               perror("server, accept failed");
               exit(1);
          }

          if (fork() == 0) {
               /* in child server */
               close(s);       /* child will use socket ns, parent uses s */
```

```
                read(ns, &buf, sizeof(buf));
                printf("Server read: %s\n", buf);
                close(ns);
                exit(0);
            }
            close(ns);           /* close socket used by child */
        }
    }
```

FILES

`/usr/include/sys/socket.h` Header file for sockets

SEE ALSO

`accept`(2), `connect`(2), `socket`(2)

NAME

`listio` – Initiates a list of I/O requests

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iosw.h>
#include <sys/listio.h>
```

`int listio (int` *cmd*`, struct listreq *`*list*`, int` *nent*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `listio` system call provides a means to initiate a list of distinct I/O requests and, optionally, waits for all of them to complete. Each I/O request in the list provides for maximum control over the desired I/O characteristics. The `listio` system call accepts the following arguments:

cmd Specifies a command. The following are valid *cmd* commands:

`LC_START` Initiates the I/O requests and returns control as soon as possible.

`LC_WAIT` Initiates the I/O requests and returns when all requests have completed.

list Points to an array of `listreq` structures. Each array includes the following members:

```
int         li_opcode;    Operation code for the request:
                          LO_READ for read and LO_WRITE
                          for write.
unsigned    li_drvr:32;   Driver dependent.
unsigned    li_flags:32;  Special request flags:  LF_LSEEK
                          to set initial file offset to li_offset.
long        li_offset;    Initial file byte offset, if LF_LSEEK
                          is set in flags.
int         li_fildes;    File descriptor (obtained from a
                          creat(2), dup(2), fcntl(2),
                          open(2), or pipe(2) system call).
char        *li_buf;      Pointer to an I/O data buffer
                          in memory.
unsigned    li_nbyte;     Number of bytes to read or write
                          for each stride.
struct iosw *li_status;   Pointer to an I/O status word
                          where the kernel will put completion
                          status for this request.  See reada(2).
int          li_signo;    Signal number of signal to send
                          to the process when the request
                          completes.  If this field is 0,
                          no signal is sent (see signal(2)
                          for a list of signal numbers).
int         li_nstride;   Number of strides; defaults to 1.
                          (On Cray MPP systems, li_nstride must
                          be 0 or 1.)
long        li_filstride; File stride in bytes;
                          default is for contiguous data flow
                          to/from the file.  (On Cray MPP systems,
                          li_filstride must be 0.)
long        li_memstride; Memory stride in bytes; default is
                          for contiguous data flow
                          to/from the memory buffer.  (On Cray MPP systems,
                          li_memstride must be 0.)
```

nent Specifies the number of requests in the list to process.

When reading or writing an *n*-dimensional array on a disk, the desired data I/O occurs at regular intervals, but it may not be contiguous. The last three variables in the `listreq` structure can be used to specify a compound request, causing multiple sections of data to be transferred. The distance from the start of one section of data on disk to the start of the next section is called the *file stride*. There is an analogous stride through memory.

When a particular request completes, the associated status word is filled in, and if `li_signo` was nonzero, the signal corresponding to the number is sent to the process. In the `iosw` structure, the status word `sw_flag` is always set upon completion; `sw_error` may contain a system call error number; and `sw_count` contains the number of bytes actually moved. For a successful compound request, `sw_count` would be `li_nstride * li_nbyte`.

The following are three ways of handling I/O completions:

- When registering for a given signal by using the `sigctl`(2) system call, the process may specify 0 rather than a handler function. After initiating one or more I/O requests with that signal number and doing any other work available, the process checks for I/O completions and, finding none, goes to sleep using the `pause`(2) system call. When the next I/O completes, the process is awakened. If an I/O completes after the process checks it but before it actually goes to sleep, the `pause`(2) system call will return immediately.
- A process may register a signal handler for a given signal and specify that signal on the `listio` request. When the I/O completes, the handler will be called and the process may service the completion. This method is interrupt or event driven. See `reada`(2) for more information about this approach. Because the kernel and library must save the process' context before calling the signal handler and restore it again after the signal handler, there is some additional overhead in this approach. The process' context is A, S, and V registers and some local memory.
- A process may specify 0, rather than a signal number, on the I/O request. In this case, the process must arrange to be awakened by some other event, perhaps a timer, or through polling the status word. The `recall`(2) (on all Cray Research system) and `recalla`(2) (on Cray PVP systems) system calls may be used to wait for ending status.

If one or more of the I/O requests are ill-formed and cannot be started, an `LC_WAIT` type command will return immediately.

NOTES

To perform a write operation, the process must be granted write permission via the security label. That is, the active security label of the process must be equal to the security label of the file.

To perform a read operation, the process must be granted read permission via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted read and write permission to the file via the security label.

RETURN VALUES

If `listio` completes successfully, a nonnegative integer is returned, indicating the number of requests that were successfully started. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `listio` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The request list is not fully contained within the process address space.
`EINTR`	A signal was caught while waiting for all I/O requests to complete during an `LC_WAIT` command.
`EINVAL`	The *cmd* argument is not a valid command.

A particular request fails if one of the following error conditions occurs:

Error Code	Description
`EAGAIN`	Mandatory file and record locking was set, `O_NDELAY` was set, and there was a blocking record lock.
`EBADF`	The `li_fildes` value is not an open file descriptor.
`EBADF`	The process is not granted read or write permission to the file via the security label and does not have appropriate privilege.
`EDEADLK`	The request was going to go to sleep and cause a deadlock situation to occur.
`EFBIG`	An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. See `ulimit`(2).
`EINTR`	An I/O request to a slow device, such as tty, was interrupted.
`EINTR`	A signal was caught waiting for an I/O quota or blocking record lock.
`EINVAL`	The `listreq` entry contains an invalid argument.
`ENOLCK`	The system record lock table was full, so the request could not go to sleep until the blocking record lock was removed.
`ENOSPC`	During a write to an ordinary file, no free space was found in the file system.
`EQACT`	A file or inode quota limit was reached for the current account ID.
`EQGRP`	A file or inode quota limit was reached for the current group ID.
`EQUSR`	A file or inode quota limit was reached for the current user ID.
`ESPIPE`	An attempt was made to specify a byte offset on a pipe or a FIFO special file (named pipe).

EXAMPLES

This example shows how to use the `listio` system call to initiate a list of input requests. The following `listio` request reads every tenth block (that is, blocks 1, 11, 21, 31, and so on) from file `datafile`. When a total of 10 data blocks is transferred, a `SIGUSR1` signal is sent to show that the transfer has been completed.

The request is initiated asynchronously (LC_START), and the data is stored in the buffer buf contiguously. Because input is performed asynchronously, the program can complete other work in parallel with the request.

```
#include <fcntl.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/iosw.h>
#include <sys/listio.h>

#define BLK_SIZ 4096

struct blk {
     char    blk_data[BLK_SIZ];
};

main()
{
     struct listreq request;
     struct iosw reqstat;
     struct blk buf[10];
     int fd;

     sigctl(SCTL_REG, SIGUSR1, 0);       /* notify process on SIGUSR1 but */
                                         /* don't execute any handler    */

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     /* Set up the I/O request */
     request.li_opcode = LO_READ;        /* request read */
     request.li_fildes = fd;             /* file descriptor */
     request.li_buf = (char *) buf;      /* store input data here */
     request.li_nbyte = BLK_SIZ;         /* each stride = 1 block */
     request.li_status = &reqstat;       /* status for this request */
     request.li_signo = SIGUSR1;         /* send signal upon completion */
     request.li_nstride = 10;            /* read 10 strides */
     request.li_filstride = 10 * BLK_SIZ;/* file stride = 10 blocks */

     sigoff();                           /* defer signal reception so SIGUSR1 */
                                         /* not received before pause() below */
     if (listio(LC_START, &request, 1) != 1) {
          perror("listio failed");
```

```
          exit(1);
     }

     /* other work can be performed here while listio request completes */

     pause();                              /* automatically calls sigon() */

     printf("Number of bytes read = %d\n\n", reqstat.sw_count);
}
```

SEE ALSO

lseek(2), pause(2), reada(2), recall(2), recalla(2), sigctl(2), signal(2), ulimit(2), write(2)

NAME

`lseek` – Moves read/write file pointer

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
```

`off_t lseek (int` *fildes*`, off_t` *offset*`, int` *whence*`);`

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `lseek` system call sets the read/write file pointer. It accepts the following arguments:

fildes Specifies a file descriptor. It is returned from a `creat`(2), `dup`(2), `fcntl`(2), or `open`(2) system call.

offset Specifies the number of bytes associated with the pointer's new location.

whence Specifies a value to indicate the pointer's location. The following are valid *whence* values.

`0` or `SEEK_SET`	Pointer is set to *offset* bytes.
`1` or `SEEK_CUR`	Pointer is set to its current location plus *offset*.
`2` or `SEEK_END`	Pointer is set to the file size plus *offset*.

Some devices such as terminals are incapable of seeking (for example, a user cannot reposition the current offset in a file to an arbitrary position). The value of the file pointer associated with such a device is undefined.

RETURN VALUES

If `lseek` completes successfully, it returns a nonnegative integer indicating the file pointer value as measured in bytes from the beginning of the file; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `lseek` system call fails and the file pointer remains unchanged if one of the following error conditions occurs:

Error Code	**Description**
`EBADF`	The *fildes* argument is not an open file descriptor.

`EINVAL` The resulting file pointer would be negative.

`EINVAL` and `SIGSYS` signal
The *whence* argument is not 0, 1, or 2.

`ESPIPE` The *fildes* argument is associated with a FIFO special file (named pipe).

FORTRAN EXTENSIONS

The `lseek` system call can be called from Fortran as a function:

```
INTEGER fildes, offset, whence, LSEEK, I
I = LSEEK (fildes, offset, whence)
```

Alternatively, `lseek` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER fildes, offset, whence
CALL LSEEK (fildes, offset, whence)
```

The Fortran program must not specify both the subroutine call and the function reference to `lseek` from the same procedure.

EXAMPLES

The following examples show different applications of the `lseek` system call. For each `lseek` example, file `datafile` is opened using the following:

```
int fd;
fd = open("datafile", O_RDWR);
```

Example 1: The following `lseek` request positions the current file pointer for `datafile` to the fourth data block (4096 bytes each) of the file:

```
lseek(fd, (long) 3 * 4096, 0);
```

Example 2: This `lseek` request returns the process's current offset into `datafile`:

```
int ret;
ret = lseek(fd, 0L, 1);
```

Example 3: This `lseek` request positions the current offset at the end of `datafile`. Therefore, the next write operation to the file will append to the file:

```
lseek(fd, 0L, 2);
```

Example 4: The following `lseek` request updates a record on `datafile`:

```
struct record recd;

read(fd, &recd, sizeof(struct record));        /* read record */
/* update record in user memory */
lseek(fd, (long) -sizeof(struct record), 1);  /* backup file offset */
write(fd, &recd, sizeof(struct record));       /* write updated record */
```

Example 5: This `lseek` request returns the current size of `datafile`. However, the current offset into the file is now positioned at the end-of-file (EOF):

```
int ret;
ret = lseek(fd, 0L, 2);
```

Example 6: This `lseek` request positions the current offset into `datafile` 100 bytes beyond the end of the file. Therefore, the next write operation to `datafile` will cause a 100-byte, 0-filled gap to be created in the file with the output data written at the current offset position.

If a `read`(2) request is issued to `datafile` while the current offset points beyond the end of the file, the request just returns an EOF condition.

```
lseek(fd, 100L, 2);
```

Example 7: This `lseek` request always fails. A file's current file pointer cannot be positioned before the beginning of a file:

```
lseek(fd, -100L, 0);
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `lseek` system call

SEE ALSO

`creat`(2), `dup`(2), `fcntl`(2), `open`(2), `read`(2)

NAME

`lsetattr` – Sets metadata for a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/vnode.h>
int lsetattr (char *fname, struct vattr *vap, int asize);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `lsetattr` system call makes the generality of the `VOP_SETATTR`() macro defined in the `sys/vnode.h` file available to user space. A single kernel call can set all the user-accessible metadata associated with a file. This information includes the time stamp, account ID, owner ID, and permission bits.

The `lsetattr` system call accepts the following arguments:

fname Points to the file name.

vap Points to an attribute structure containing the desired metadata changes.

asize Contains the size of the structure pointed to by the *vap* argument. *asize* provides robustness across minor changes in the attribute structure definition; it will not cause an error return.

The `lsetattr` call does not follow symbolic links.

NOTES

Since some attribute change requests are validated above the vnode switch and since `lsetattr` goes directly to the vnode switch, the initial implementation is restricted. A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to set metadata for a file.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to set metadata for a file.

RETURN VALUES

If `lsetattr` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error. For an explanation of the error, enter an `explain`(1) command or see the error code list on the `intro`(2) man page.

These calls are referenced on this man page instead of listing the errors in an ERRORS section because `lsetattr` can behave like one or more system calls depending upon the arguments specified and can cause several different errors.

EXAMPLES

The following examples illustrate different uses of the `lsetattr` system call.

Example 1: The following code fragment shows how a single `lsetattr` system call can perform the function of combined `chown`(2) and `chmod`(2) system calls.

```
vap->va_uid = 1005;
vap->va_mode = 0644;
vap->va_mask = AT_UID | AT_MODE;
ex = lsetattr( "file_name", vap, sizeof(*vap));
if (ex < 0) fprintf(stderr, "example failed\n");
```

Example 2: This example shows how to set the sitebits on a symbolic link.

```
vap->va_sitebits = SITE_BITS_VALUE;
vap->va_mask = AT_SITEBITS;
ex = lsetattr( "link_name", vap, sizeof(*vap));
if (ex < 0) fprintf(stderr, "second example also failed\n");
```

Example 3: A final example illustrates how to set the account ID.

```
vap->va_acid = 42;
vap->va_mask = AT_ACID;
ex = lsetattr( "nuther_file", vap, sizeof(*vap));
if (ex < 0) fprintf(stderr, "third example busted, too\n");
```

FILES

`/usr/include/types.h`	Contains types required by ANSI X3J11
`/usr/include/vnode.h`	Contains vnode, attribute structure, and bit definitions for `va_mask`.

SEE ALSO

`chmod`(2), `chown`(2), `intro`(2)

`explain`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`mkdir` – Makes a directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir (const char *path, mode_t mode);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `mkdir` system call creates a new directory. It accepts the following arguments:

path Names the new directory.

mode Provides the mode of the new directory. The protection part of the *mode* argument is modified by the process' mode mask (see `umask`(2)).

The owner ID of the new directory is set to the process' real user ID. The group ID of the new directory is set to the group ID of the parent directory. The newly created directory is empty, with the exception of entries for "." and "..".

NOTES

The new directory is assigned the active security label of the process.

The active security label of the process must fall within the security label range of the file system in which the directory is being created.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

The process must be granted write permission to the parent directory via the security label.

To be granted write permission to the parent directory, the active security label of the process must equal the security label of the parent directory.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.

PRIV_DAC_OVERRIDE	The process is granted write permission to the parent directory via the permission bits and access control list.
PRIV_MAC_READ	The calling process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_WRITE	The process is granted write permission to the parent directory via the security label.

If the PRIV_SU configuration option is enabled, to every component of the path prefix. If the PRIV_SU configuration option is enabled, the super user is granted write permission to the parent directory.

RETURN VALUES

If mkdir completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The mkdir system call fails and no directory is created if one of the following error conditions occurs:

Error Code	Description
EACCES	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
EACCES	Parent directory label is not equal to the active security label of the process.
EEXIST	The specified file already exists.
EFAULT	The *path* argument points outside the allocated address space of the process.
EFLNEQ	Attempt was made to create a directory outside the bounds of the file system.
EFLNEQ	The active security label of the process falls outside the range of the file system.
EIO	An I/O error has occurred during access of the file system.
EMLINK	The maximum number of links to the parent directory would be exceeded.
ENOENT	A component of the path prefix does not exist.
ENOENT	The path is longer than the maximum allowed.
ENOTDIR	A component of the path prefix is not a directory.
EQACT	A file or inode quota limit was reached for the current account ID.
EQGRP	A file or inode quota limit was reached for the current group ID.
EQUSR	A file or inode quota limit was reached for the current user ID.
EROFS	The path prefix resides on a read-only file system.

SEE ALSO

`umask`(2)

NAME

`mknod`, `mkfifo` – Makes a directory or a special or regular file

SYNOPSIS

```
#include <unistd.h>
```

`int mknod (char *`*path*`, int` *mode*`, int` *dev*`, long` *p0*`, long` *p1*`,...,`*p7*`);`

```
#include <sys/types.h>
#include <sys/stat.h>
```

`int mkfifo (const char *`*path*`, mode_t` *mode*`);`

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `mkfifo`)

DESCRIPTION

The `mknod` system call creates a file or directory. It accepts the following arguments:

path Names the new file or directory.

mode Specifies the mode of the new file. (Symbolic names for these constants exist in `<sys/stat.h>`). The following are valid values for *mode*.

0170000 File type, as follows:

0010000	FIFO special file (named pipe)
0020000	Character special file
0040000	Directory
0060000	Block special file
0100000	Regular file
0120000	Offline file without data
0110000	Offline file with data

0004000 Set user ID on execution

0002000 Set group ID on execution

0000777 Access permissions, as follows:

0000400	Read by owner
0000200	Write by owner

0000100	Execute (search on directory) by owner
0000070	Read, write, or execute (search) by group
0000007	Read, write, or execute (search) by others

dev Specifies a device.

If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device.

If *mode* does not indicate a block special or character special device, *dev* is ignored.

p0, *p1*, . . ., *p7* Specifies device-specific parameter words. These words give the operating system more information about the device's configuration.

The file's owner ID is set to the effective user ID of the process. The file's group ID is set to the group ID of the parent directory.

Values of *mode* other than the preceding are undefined and should not be used. The low-order 9 bits of *mode* are modified by the file mode creation mask of the process; all bits set in the mask are cleared. See `umask`(2).

Only a process with appropriate privilege can use this system call.

The file handle for an offline file created by the `mknod` system call has zero elements.

The `mkfifo` routine creates a new FIFO special file named by the path name to which *path* points. The file permission bits of the new FIFO are initialized from *mode*. The low-order 9 bits of *mode* are modified by the file mode creation mask of the process; all bits set in the mask are cleared. See `umask`(2). `mkfifo` sets the file's owner ID and group ID, following the same rules that `mknod` uses.

NOTES

The active security label of the calling process must fall within the security label range of the file system on which the new node will reside.

If the `FSETID_RESTRICT` option is enabled, only a process with appropriate privilege can create set-user-ID or set-group-ID files.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the parent directory, the active security label of the process must equal the security label of the directory.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_ADMIN`	The process is allowed to use this system call.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.

PRIV_DAC_OVERRIDE	The process is granted write permission to the parent directory via the permission bits and access control list.
PRIV_FSETID	When the FSETID_RESTRICT option is enabled, the process is allowed to create set-user-ID or set-group-ID files.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.
PRIV_MAC_WRITE	The process is granted write permission to the parent directory via the security label.
PRIV_RESTART	The process is allowed to create a restart file.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_MKNOD permbit is allowed to use this system call. The super user is granted search permission to every component of the path prefix and is granted write permission to the parent directory. The super user is allowed to create restart files. If the PRIV_SU and FSETID_RESTRICT configuration options are enabled, the super user is allowed to create set-user-ID and set-group-ID files.

RETURN VALUES

If mknod or mkfifo completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The mknod or mkfifo system call fails and the new file is not created if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
EACCES	Write permission is denied to the parent directory.
EEXIST	The specified file exists.
EFAULT	The *path* argument points outside the allocated process address space.
EFLNEQ	The active security label of the calling process does not fall within the range of the file system on which the new file or directory will reside.
EINVAL	The call contains an argument that is not valid. For example, an attempt was made to create a file or directory on a symbolic link.
ENOENT	A component of the path prefix does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The process does not have appropriate privilege to use this system call.
EROFS	The directory in which the file is to be created is located on a read-only file system.

FORTRAN EXTENSIONS

The `mknod` system call can be called from Fortran as a function:

```
CHARACTER*n path
INTEGER  mode, dev, MKNOD, I
I = MKNOD (path, mode, dev)
```

Alternatively, `mknod` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
INTEGER mode, dev
CALL MKNOD (path, mode, dev)
```

The Fortran program must not specify both the subroutine call and the function reference to `mknod` from the same procedure. On all Cray PVP systems except CRAY T90 systems, *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte.

EXAMPLES

The following examples illustrate use of the `mkfifo` and `mknod` system calls.

Example 1: This `mkfifo` request creates a named pipe (that is, FIFO special file), called `name_pipe` with the specified permissions:

```
if (mkfifo("name_pipe", 0640) == -1) {
     perror("mkfifo failed");
     exit(1);
}
```

Example 2: This example shows one of the applications for the `mknod` system call, the creation of a named pipe (that is, FIFO special file). This `mknod` request creates a named pipe called `name_pipe` with the specified permissions.

```
if (mknod("name_pipe", 010640) == -1) {
     perror("mknod failed");
     exit(1);
}
```

FILES

`/usr/include/sys/stat.h`	Contains ANSI C prototype for the `mkfifo` system call
`/usr/include/sys/types.h`	Contains data type definitions and definition for `mode_t`
`/usr/include/unistd.h`	Contains C prototype for the `mknod` system call

SEE ALSO

`chmod`(2), `creat`(2), `exec`(2), `umask`(2)

`mkdir`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`fs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`mount` – Mounts a file system

SYNOPSIS

`#include <sys/mount.h>`

`int mount (char *`*spec*`, char *`*fsname*`, char *`*dir*`, char *`*options*`, int` *flags*`,`
`int` *fstyp*`, char *`*data*`, int` *datalen*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `mount` system call requests that a removable file system contained on a block or character special file be mounted on a directory as specified by the following arguments:

spec Points to the block or character special file's path name. The *spec* argument may be a block special file.

fsname Points to the file system full path name.

dir Points to the path name of the directory on which *spec* is to be mounted. On successful completion, references to the *dir* file refer to the root directory on the mounted file system. The full path must be specified.

options Points to the file system mount options. This is a comma-separated list of words from the `-o` option on the `mount`(8) command.

flags Identifies the flag whose low-order bit controls write permission on the mounted file system. If the bit is 1, writing is forbidden; otherwise, writing is permitted according to individual file accessibility. The *flags* argument can contain three values, as defined in `/usr/include/sys/mount.h`:

- `MS_RDONLY` Read only.
- `MS_FSS` The *fstyp* argument has meaning.
- `MS_DATA` The *data* and *datalen* arguments have meaning. This flag is used by the NFS file system type.

fstyp Specifies the file system type being mounted (see `sysfs`(2)). This field is interpreted only if the `MS_FSS` flag is set.

data Points to file-system-specific mount information of size *datalen*. The *data* and *datalen* arguments are interpreted only if `MS_DATA` is set in *flags*.

datalen Specifies size of mount information, as described in explanation of *data* pointer.

Only an appropriately privileged process can use this system call.

NOTES

If the MLS_OBJ_RANGES configuration is enabled, the minimum and maximum security levels (with the exception of the syslow and/or syshigh security levels) and the authorized compartments of a file system must fall within the authorized ranges of the UNICOS system, otherwise the mount request fails.

File systems that have not been explicitly assigned a security label range (by using the labelit(8) or mkfs(8) commands) are considered to have the security label range [level 0: none, level 0: none].

The mount point label of the file system must be less than or equal to the lowest label assigned to the file system.

When the MLS_OBJ_RANGES option is set to SECURE, the security label range of the file system must fall within the security label range of the UNICOS system.

To mount a file system with DEV_ENFORCE_ON set to ON, the device must be off or the minimum and maximum security level of the file system must be within the minimum and maximum security levels authorized for the device. Also, the authorized security compartments for the file system must be equal to or a subset of the authorized compartments for the device, and the device must be labeled as multilevel.

A process is granted search permission to a component of the path prefix only if the active security label of the process is greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_ADMIN	The process is allowed to use this system call.
PRIV_DAC_OVERRIDE	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
PRIV_MAC_READ	The process is granted search permission to every component of the path prefix via the security label.

If the PRIV_SU configuration option is enabled, a super user is allowed to use this system call and is granted search permission to every component of the path prefix.

RETURN VALUES

If mount completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The mount system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.

`EBUSY`	The device associated with *spec* is currently mounted.
`EBUSY`	The *dir* argument is currently mounted, is someone's current working directory, or is otherwise busy.
`EFAULT`	The *spec* or *dir* argument points outside the allocated process address space.
`EINVAL`	The *fstyp* argument is not valid.
`ENODEV`	The file system super-block device number does not match the block special device.
`ENOENT`	Any of the specified files does not exist.
`ENOEXEC`	The super block or dynamic block of the file system is corrupt.
`ENOTDIR`	A component of a path prefix is not a directory.
`ENOTDIR`	The *dir* argument is not a directory.
`ENXIO`	The device associated with *spec* does not exist.
`EPERM`	The process does not have appropriate privilege to use this system call.
`ESYSLV`	The upper security level of the file system is greater than the upper security level of the UNICOS system and is not the `syshigh` security label.
`ESYSLV`	The lower security level of the file system is less than the lower security level of the UNICOS system and is not the `syslow` security label.
`ESYSLV`	A user with an active security level (nonzero) tried to mount a non-UNICOS file system. This file system is treated as unclassified with lower and upper security levels equal to 0.
`ESYSLV`	The `MLS_OBJ_RANGES` option is enabled, and the authorized compartments of the file system are not a subset of the authorized compartments for the UNICOS system.
`ESYSLV`	The `DEV_ENFORCE_ON` option is enabled and the file system label is not within the device label range.
`ESYSLV`	The `DEV_ENFORCE_ON` option is enabled and the device is on, but not labeled as multilevel.

FILES

`/usr/include/sys/mount.h`	Mount structure

SEE ALSO

`sysfs`(2), `umount`(2)

`libudb`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`labelit`(8), `mkfs`(8), `mount`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`msgctl` – Provides message control operations

SYNOPSIS

```
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `msgctl` system call provides a variety of message control operations. It accepts the following arguments:

msqid Specifies a message queue identifier.

cmd Specifies a message control operation. The following are valid *cmd* values.

`IPC_STAT` Places the current value of each member of the `msqid_ds` data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in the `sys/msg.h` include file (see `msg`(5)). The calling process must have read permission to the message queue associated with *msqid*.

`IPC_SET` Sets the value of the following members of the `msqid_ds` data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*.

```
msg_perm.uid
msg_perm.gid
msg_perm.mode     /* only access permission bits */
msg_qbytes
```

`IPC_SET` can be executed only by a process that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msqid*. Only a process with the appropriate privilege can raise the value of `msg_qbytes`.

`IPC_RMID` Removes the message queue identifier specified by *msqid* from the system and destroys the message queue and `msqid_ds` data structure associated with it. `IPC_RMID` can be executed only by a process that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msqid*.

`IPC_SETACL` (Secure systems only) Sets the access control list (ACL) on the message queue specified by *msqid*. The `ipc_perm` structure within the `msqid_ds` structure pointed to by *buf* contains a pointer, `ipc_acl`, to an `acl_rec` structure with the required ACL entries, and a count of those entries, `ipc_aclcount`. If an ACL exists for the message queue, it is replaced by the one provided with this call. If `ipc_aclcount` is 0, any existing ACL is removed. The calling process must be the owner of the message queue specified by *msqid*.

`IPC_GETACL` (Secure systems only) Retrieves the access control list (ACL) for the message queue specified by *msqid*. The `ipc_perm` structure within the `msqid_ds` structure pointed to by *buf* contains a pointer, `ipc_acl`, to an `acl_rec` structure where the ACL entries are to be returned. The count of entries to be returned is specified in the `ipc_aclcount` field. If there are more than `ipc_aclcount` entries, only the first `ipc_aclcount` entry is returned. If there are fewer than `ipc_aclcount` entries, all entries are returned. The return value indicates the number of entries returned. If there is no ACL, the return value is 0. The calling process must have read permission to the message queue specified by *msqid*.

`IPC_SETLABEL` (Secure systems only) Sets the security label on the message queue specified by *msqid*. The `ipc_perm` structure within the `msqid_ds` structure pointed to by *buf* contains a security level, `ipc_slevel`, and a compartment set, `ipc_scomps`, to be set in the security label on the message queue. Only a process with the appropriate privilege can set the security label of a message queue.

buf Points to a structure.

NOTES

A process is granted read permission to a message queue only if the active security label of the process is greater than or equal to the security label of the message queue, and the process is granted read access by the message queue access control list (ACL) (if one is assigned). This applies to the `IPC_STAT` and `IPC_GETACL` operations.

The `IPC_SET`, `IPC_RMID`, and `IPC_SETACL` operations require that the active security label of the process is equal to the security label of the message queue.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_READ`	The process is considered to meet the security label requirements for being granted read permission to a message queue.
`PRIV_MAC_WRITE`	The process is considered to meet the security label requirements for performing an `IPC_SET`, `IPC_RMID`, or `IPC_SETACL` operation.

PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read permission to a message queue.
PRIV_FOWNER	The process is considered to meet the message queue ownership requirements for the IPC_SET, IPC_RMID, and IPC_SETACL operations. For the IPC_SET operation, the process is also permitted to raise the value of msg_qbytes.
PRIV_MAC_UPGRADE	The process is allowed to raise the security label of a message queue.
PRIV_MAC_DOWNGRADE	The process is allowed to lower the security label of a message queue.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown previously. The super user is considered the owner of a message queue, and is granted read permission to that message queue.

RETURN VALUES

If msgctl completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The msgctl system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	The *cmd* argument is IPC_STAT, and the calling process does not have read permission (see msg(5)).
EACCES	The *cmd* argument is IPC_GETACL, and the calling process does not have read permission.
EFAULT	The *buf* argument points to an illegal address.
EFAULT	The *cmd* argument is IPC_SETACL or IPC_GETACL and the ipc_acl field in *buf* points to an illegal address.
EINVAL	The *msqid* argument is not a valid message queue identifier.
EINVAL	The *cmd* argument is not a valid command.
EINVAL	The *cmd* argument is IPC_SET, and msg_perm.uid or msg_perm.gid is not valid.
EINVAL	The *cmd* argument is IPC_SETACL, and one of the following is true: • The ipc_aclcount field in *buf* is 0, but there is no ACL associated with *msqid*. • The ipc_aclcount field in *buf* is less than 0 or greater than 256. • The ACL supplied failed validation.

ENOMEM	The *cmd* argument is IPC_SETACL, and no memory is available to store the ACL. The command should be retried at a later time.
EPERM	The *cmd* argument is IPC_RMID or IPC_SET, and the effective user ID of the calling process is not equal to the value of msg_perm.cuid or msg_perm.uid in the msgid_ds data structure associated with *msqid*; the calling process does not have the appropriate privilege.
EPERM	The *cmd* argument is IPC_SET, an attempt is being made to increase to the value of msg_qbytes, and the calling process does not have the appropriate privilege.
EPERM	The *cmd* argument is IPC_SETLABEL, and the calling process does not have the appropriate privilege.
EPERM	The *cmd* argument is IPC_SETACL, and the calling process does not meet ownership requirements and does not have the appropriate privilege.

FILES

/usr/include/sys/msg.h Contains message-related data structures and macros

SEE ALSO

msgget(2), msgrcv(2), msgsnd(2)

ipc(5), msg(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

`msgget` – Accesses the message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgget (key_t key, int msgflg);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `msgget` system call returns the message queue identifier. It accepts the following arguments:

key Specifies the message queue.

msgflg Specifies a flag value.

A message queue identifier and associated message queue and data structure (see `msg`(5)) are created for *key* if one of the following is true:

- *key* is `IPC_PRIVATE`.
- *key* does not already have a message queue identifier associated with it, and the value of *msgflg*&`IPC_CREAT` is not 0.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` are set to the effective user ID and effective group ID, respectively, of the calling process.
- The access permission bits of `msg_perm.mode` are set to the access permission bits of *msgflg*.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

NOTES

If the calling process has the `ipc_persist` permission bit, the message queue is created as a persistent queue. Persistent message queues will not be removed from the system unless a `msgctl`(2) system call with the command `IPC_RMID`, or an `ipcrm`(1) command, is performed on the queue.

If the calling process does not have this permission bit, the message queue is linked into a list of nonpersistent queues belonging to the session of which the process is a member. When the last process of the session terminates, all the message queues linked to the session are removed from the system.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_RESOURCE`	The process is considered to have the `ipc_persist` permission bit.

If the `PRIV_SU` configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is considered to have the `ipc_persist` permission bit.

RETURN VALUES

If `msgget` completes successfully, a nonnegative integer, namely a message queue identifier, is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `msgget` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A message queue identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted (see `ipc`(7)).
`EEXIST`	A message queue identifier exists for *key* but the values of *msgflg*&`IPC_CREAT` and *msgflg*&`IPC_EXCL` are both nonzero.
`ENOENT`	A message queue identifier does not exist for *key*, and the value of *msgflg*&`IPC_CREAT` is 0.
`ENOSPC`	A message queue identifier is to be created, but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

FILES

`/usr/include/sys/msg.h`	Contains message-related data structures and macros

SEE ALSO

`msgctl`(2), `msgrcv`(2), `msgsnd`(2)

`ipcrm`(1), `ipcs`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`stdipc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`ipc`(5), `msg`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`msgrcv` – Reads a message from a message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `msgrcv` system call reads a message from the queue associated with the message queue identifier and places it in the user-defined buffer. It accepts the following arguments:

msqid Specifies a message queue identifier.

msgp Points to a user-defined buffer.

This user-defined buffer must contain a message type field (of type `long int`) followed by the data portion for the message text. The following structure is defined in the include file `sys/msg.h` (see `msg`(5)):

```
struct msgbuf {
      long int    msgtype;    /* message type */
      char        msgtext[1]; /* message text */
}
```

The structure member `msgtype` is the received message's type, as specified by the sending process. The structure member `msgtext` is the text of the message.

msgsz Specifies the size, in bytes, of `msgtext`. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and the value of *msgflg*&`MSG_NOERROR` is not 0. The truncated part of the message is lost; no indication of the truncation is given to the calling process.

msgtyp Specifies the type of message requested. The following are valid types.

- If *msgtyp* is 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg Specifies a flag value.
If a message of the desired type is not on the queue, *msgflg* identifies one of the following actions:

- If the value of *msgflg*&IPC_NOWAIT is not 0, the calling process returns immediately with a return value of –1 and sets errno to ENOMSG.
- If the value of *msgflg*&IPC_NOWAIT is 0, the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier *msqid* is removed from the system. When this occurs, errno is set to EIDRM, and a value of –1 is returned.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes execution in the manner prescribed in sigaction(2).

Upon successful completion, the data structure associated with *msqid* (see msg(5)) is changed as follows:

- msg_qnum is decremented by 1.
- msg_lrpid is set to the process ID of the calling process.
- msg_rtime is set to the current time.

NOTES

A process is granted read permission to a message queue only if the active security label of the process is greater than or equal to the security label of the message queue, and the process is granted read access by the message queue access control list (ACL) (if one is assigned).

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_MAC_READ	The process is considered to meet the security label requirements for being granted read permission to a message queue.
PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read permission to a message queue.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is granted read permission to a message queue.

RETURN VALUES

If msgrcv completes successfully, a value that indicates the number of bytes actually placed in the *msgtext* buffer is returned. Otherwise, a value of –1 is returned, and errno is set to indicate the error; no message is received.

ERRORS

The msgrcv system call fails and receives no message if one of the following error conditions occurs:

Error Code	Description
E2BIG	The value of *msgtext* is greater than *msgsz*, and *msgflg*&MSG_NOERROR is equal to 0.
EACCES	Operation permission is denied to the calling process (see msg(5)).
EFAULT	*msgp* points to an illegal address.
EIDRM	The message queue identifier *msqid* is removed from the system.
EINTR	The msgrcv system call was interrupted by a signal.
EINVAL	*msqid* is not a valid message queue identifier.
EINVAL	*msgsz* is less than 0.
ENOMSG	The queue does not contain a message of the desired type, and *msgtyp*&IPC_NOWAIT is not 0.

FILES

/usr/include/sys/msg.h	Contains message-related data structures and macros

SEE ALSO

msgctl(2), msgget(2), msgsnd(2), sigaction(2)

ipc(5), msg(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ipc(7) Online only

NAME

`msgsnd` – Sends a message to a message queue

SYNOPSIS

`#include <sys/msg.h>`

`int msgsnd (int` *msqid*`, const void *`*msgp*`, size_t` *msgsz*`, int` *msgflg*`);`

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `msgsnd` system call sends a message to the queue associated with the message queue identifier. It accepts the following arguments:

msqid Specifies a message queue identifier.

msgp Points to a user-defined buffer. It must contain a message type field (of type `long int`) followed by a data portion for the message text. The following structure is defined in the include file `sys/msg.h` (see `msg`(5)):

```
struct msgbuf {
      long msgtype;    /* message type */
      char msgtext[];  /* message text */
}
```

The structure member `msgtype` is a positive integer that can be used by the receiving process for message selection.

msgsz Specifies the length of `msgtext` in the number bytes. *msgsz* can range from 0 to a system-imposed maximum.

msgflg Specifies the action to be taken if the message cannot be immediately processed. *msgflg* specifies the action to be taken if one or more of the following are true and thus prevents the message from being immediately processed:

- The number of bytes already on the queue is equal to `msg_qbytes` (see `msg`(5)).
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

The following actions are available:

- If *msgflg*&`IPC_NOWAIT` is not 0, the message is not sent and the calling process returns immediately.

- If *msgflg*&`IPC_NOWAIT` is 0, the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists. In this case, the message is sent.
  - The message queue identifier *msqid* is removed from the system (see `msgctl`(2)). When this occurs, `errno` is set to `EIDRM`, and a value of –1 is returned.
  - The calling process receives a signal that is to be caught. In this case, the message is not sent and the calling process resumes execution in the manner prescribed in the `sigaction`(2) system call.

Upon successful completion, the data structure associated with *msqid* (see `msg`(5)) is changed as follows:

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set to the process ID of the calling process.
- `msg_stime` is set to the current time.

NOTES

A process is granted write permission to a message queue only if the active security label of the process is equal to the security label of the message queue, and the process is granted write access by the message queue access control list (ACL) (if one is assigned).

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_WRITE`	The process is considered to meet the security label requirements for being granted write permission to a message queue.
`PRIV_DAC_OVERRIDE`	The process is considered to meet the permission mode and ACL requirements for being granted write permission to a message queue.

If the `PRIV_SU` configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is granted write permission to a message queue.

RETURN VALUES

If `msgsnd` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error, and no message is sent.

ERRORS

The `msgsnd` system call fails and sends no message if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Operation permission is denied to the calling process (see `ipc`(7)).

`EAGAIN` The message cannot be sent for one of the reasons cited in the DESCRIPTION section, and *msgflg*&`IPC_NOWAIT` is not 0.

`EFAULT` *msgp* points to an illegal address.

`EIDRM` The message queue identifier *msqid* is removed from the system.

`EINTR` The `msgsnd` system call was interrupted by a signal.

`EINVAL` The value of *msqid* is not a valid message queue identifier.

`EINVAL` The value of *msgtype* is less than 1.

`EINVAL` The value of *msgsz* is less than 0 or greater than the system-imposed limit.

FILES

`/usr/include/sys/msg.h` Contains message-related data structures and macros

SEE ALSO

`msgctl`(2), `msgget`(2), `msgrcv`(2), `sigaction`(2)

`ipc`(5), `msg`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`mtimes` – Provides multitasking execution overlap profile

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mtimes.h>

struct mtms *mtimes (struct mtms *buf);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `mtimes` system call lets users have a structure in user memory continually updated with multitasking execution overlap information. (This is the same information that appears in accounting records for multitasking programs.) From this `mtms` structure, users can determine how much execution time the multitasking program has accumulated in an interval.

The `mtimes` system call accepts the following argument:

buf Specifies the address of the structure to receive the data. If the address is 0, the structure will no longer be updated.

The `mtms` structure contains the following members:

```
time_t   mtms_update;              /* Time of last update */
short    mtms_conn;                /* # of cpus presently connected */
time_t   mtms_mutime[NCPU];        /* Multitask cpu utilization */
```

Update of the structure stops if one of the following occurs:

- The process shrinks, placing the structure outside the program's address space.
- An `exec`(2) system call is executed.

NOTES

The times in the `mtms` structure refer to the period since the multitasking group began execution, not to the period since the invocation of the `mtimes` call.

Monitoring the `mtms` structure at the user level is somewhat tricky because the operating system running in another CPU may be updating it at the same time.

RETURN VALUES

The `mtimes` system call returns the address of the `mtms` structure. If the value returned is 0, the feature is disabled; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `mtimes` system call fails if the following error condition occurs:

Error Code	**Description**
`EFAULT`	The *buf* argument points outside the program's address space.

SEE ALSO

`exec`(2)

`MTIMESX`(3F), `MTTIMES`(3F) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`SECOND`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`newarraysess` – Starts a new array session

SYNOPSIS

```
#include <unistd.h>
int newarraysess (void);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The `newarraysess` system call creates a new array session and moves the current process from its original array session to the new one. The parents, children, and siblings of the current process are not affected by this move and remain in their original array sessions.

The system generates a handle for the new array session. Normally, the new handle is guaranteed to be unique on the current system only, although some systems may be able to assign global array session handles that are unique across an entire array of systems by setting the `asmachid` system variable. Otherwise, the range of values that the system may assign for array session handles is defined by the system variables `minash` and `maxash`. If necessary, the `setash`(2) system call can be used to override the default handle after the array session has been created.

Ordinarily, a new array session should be started whenever the conceptual equivalent of a login is performed. This includes programs that do conventional logins (for example, `login`(1) or `telnet`(1B)), as well as programs that are essentially logging in to do work on behalf of another user, such as `cron`(8) or batch queueing systems.

RETURN VALUES

If `newarraysess` completes successfully, a value of 0 is returned; otherwise, a value of `-1` is returned, and `errno` is set to indicate the error.

ERRORS

The `newarraysess` system call fails if the following condition occurs:

Error Code	**Description**
`ENOMEM`	The system is unable to allocate memory or other resources for the new array session.

SEE ALSO

`setash`(2)

`login`(1), `telnet`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`array_services`(7), `array_sessions`(7),

`cron`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

nice, nicem – Changes priority of processes

SYNOPSIS

```
#include <unistd.h>

int nice (int incr);

#include <sys/category.h>
#include <unistd.h>

int nicem (int category, int id, int incr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `nice`)

DESCRIPTION

The `nice` system call adds a specified value to the nice value of the calling process. A process' nice value is a positive number for which a greater value results in lower CPU priority. Only an appropriately privileged process can specify a negative *incr*.

Only an appropriately privileged process can set the nice value for a process that it does not own.

The system imposes a maximum nice value of 39 and a minimum nice value of 0. When values above or below these limits are requested, the nice value will be set to the corresponding limit.

The `nicem` system call changes the nice value of a process or group of processes as specified by the following arguments:

category Specifies a category. The following are valid values for *category*: `C_PROC`, `C_PGRP`, `C_JOB`, or `C_UID`.

id Specifies the *pid*, *pgrp*, *jid*, or *uid* corresponding to *category*. A *pid* of 0 means the current process, a *pgrp* of 0 means the current process group, a *jid* of 0 means the current job, and a *uid* of 0 means the current user.

incr Specifies the value to be added to the nice value of the calling process.

NOTES

The active security label of the process must be greater than or equal to the security label of every affected process.

To set a nice value, the active security label of the process must equal the security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_MAC_READ`	The active security label of the process is considered to be greater than or equal to the security label of all affected processes.
`PRIV_MAC_WRITE`	The active security label of the process is considered to equal the security label of all affected processes.
`PRIV_POWNER`	The process is considered the owner of all affected processes.
`PRIV_RESOURCE`	The process is allowed to specify a negative value for *incr*.

If the `PRIV_SU` configuration option is enabled, and is allowed to specify a negative value for *incr*. If the `PRIV_SU` configuration option is enabled, the super user overrides all security label restrictions.

RETURN VALUES

When `nice` completes successfully, it returns the new nice value minus 20 (–20 <= return <= 19), and `errno` is never set. When `nicem` completes successfully, it returns the new nice value unchanged (0 <= return <= 39). Otherwise a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `nice` or `nicem` system call fails if one of the following error conditions occurs:

Error Code	Description
`EINVAL`	One of the arguments contains an invalid value.
`EPERM`	The process specified a negative value for *incr* and does not have appropriate privilege.
`EPERM`	The process does not own all affected processes and does not have appropriate privilege.
`ESRCH`	No process can be found to match the *category* and *id* requests.

FORTRAN EXTENSIONS

The `nice` system call can be called from Fortran as a function:

```
INTEGER incr, NICE, I
I = NICE (incr)
```

The `nicem` system call can be called from Fortran as a function:

```
INTEGER category, id, incr, NICEM, I
I = NICEM (category, id, incr)
```

EXAMPLES

This example illustrates how the `nice` and `nicem` system calls can change the nice value of a process and thereby affect the CPU priority of the process. The following `nice` and `nicem` requests return the current nice value of the calling process and increase the nice value to lower the CPU priority of the process. The current nice value returned by `nice` is offset by a factor of –20.

Only users with super-user status can lower the nice value of a process and thereby raise the CPU priority.

```
#include <sys/category.h>
#include <unistd.h>

main()
{
     int i;

     printf("Current nice value from nice(2) = %d\n", nice(0));
     printf("Current nice value from nicem(2) = %d\n", nicem(C_PROC, 0, 0));

     if ((i = nicem(C_PROC, 0, 5)) == -1)
          perror("nicem (+incr) failed");
     else
          printf("New nice value = %d\n", i);

     if ((i = nicem(C_PROC, 0, -5)) == -1)     /* for non-superusers - */
          perror("nicem (-incr) failed");      /* always fails */
     else
          printf("New nice value = %d\n", i);
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `nice` and `nicem` system calls

SEE ALSO

`exec`(2)

`nice`(1), `renice`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`passwd`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`nsecctl` – Accesses or manipulates network security information

SYNOPSIS

```
#include <net/nsecctl.h>
#include <net/al.h>
```

`int nsecctl (int` *op*`, caddr_t` *entry*`, struct al_addr` **addr_list*`, int` *num*`);`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `nsecctl` system call manipulates and accesses network security information stored in the kernel.

It accepts the following arguments:

op Specifies the operation to be performed. A version number is built into the upper halfword of the operation. The following operations are valid:

`NALM_ADD`	Adds one or more network access list (NAL) entries.
`NALM_DELETE`	Deletes a NAL entry.
`NALM_RESOLVE`	Gets a NAL entry.
`WALM_ADD`	Adds one or more workstation access list (WAL) entries.
`WALM_DELETE`	Deletes a WAL entry.
`WALM_RESOLVE`	Gets a WAL entry.
`WALM_CHECK`	Checks the access for a given WAL entry.
`MAPM_ADD`	Adds a map.
`MAPM_DELETE`	Deletes a map.

entry Specifies the address of a structure, the type of which depends on *op*. For NAL operations, this is a pointer to a `struct nalentry`. For WAL operations, this is a pointer to a `struct walentry`. For map operations, this is a pointer to a `struct ipso_map`.

On calls in which a `walentry` or `nalentry` is expected back, the resultant information is placed at this address.

addr_list Specifies a pointer to an array of `al_addr` structures. These specify either the host or network addresses described by the NAL or WAL entry. This argument is ignored for map operations.

For NAL and WAL operations, this array is copied to the calling process upon completion of the operation. If an error occurs in an add or delete operation for a NAL or a WAL entry, the error number appears in the upper halfword of the `al_flags` field for that element of the `al_addr` list. More than one element of the list can result in an error. Elements of the list for which an error does not appear have been processed (added or deleted) successfully, except in cases where `errno` is set to `EACCES`, `ENOBUFS`, or `EINVAL`.

num Specifies the number of entries in the `al_addr` list. This argument is ignored for map operations.

NOTES

The calling process must have `PRIV_ADMIN` effective privilege.

RETURN VALUES

If `nsecctl` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

If all elements of NAL and WAL multiple address requests are processed successfully, a value of 0 is returned. If an error occurs for any element, a value of -1 is returned, and `errno` is set to indicate the last error encountered.

ERRORS

The `nsecctl` system call fails if one of the following error conditions occurs:

Error code	**Description**
`EACCES`	The calling process did not have `PRIV_ADMIN` effective privilege. No list elements were processed.
`EEXIST`	An attempt was made to add an entry that already exists.
`EINVAL`	The argument specified was not valid, or the calling process was not built with the correct version of `nsecctl.h`. No list elements were processed.
`ENOBUFS`	No buffer space is available. No list elements were processed.
`ESRCH`	An attempt was made to delete or to resolve an entry that does not exist.

For an `nsecctl` call involving multiple addresses, each element that results in an error is processed if possible, or else has the error number in the upper halfword of the `al_flags` field for that element of the `al_addr` list.

FILES

usr/include/net/al.h	Header file for address list al_addr structure
/usr/include/net/map.h	Header file for ipso_map structure
/usr/include/net/nal.h	Header file for nalentry structure
/usr/include/net/nsecctl.h	Header file for nsecctl requests
/usr/include/net/wal.h	Header file for walentry structure

SEE ALSO

spnet(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

open – Opens a file for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (const char *path, int oflag [,mode_t mode] [,long cbits]
[,int cblks]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The open system call opens a file descriptor and sets the file status flags according to the value of the following arguments:

path Points to a path name of a file.

oflag Specifies status flags. These are constructed by flags from the following list. Only one of the first three flags may be used.

O_RDONLY If set, opens file for reading only.

O_WRONLY If set, opens file for writing only. To append a file, see the O_APPEND flag.

O_RDWR If set, opens file for reading and writing.

O_RAW If set, reads or writes whole sectors of data into user space, bypassing system buffers. Usually, the system does automatic read-ahead and write-behind to improve performance. Use of O_RAW does not imply O_LDRAW.

O_LDRAW When used in conjunction with O_RAW, I/O bypasses logical device cache as well as system buffer cache. Use of O_LDRAW does not imply O_RAW.

O_NDELAY If set, affects subsequent reads and writes (see read(2) and write(2)).

When opening a FIFO special file (named pipe) with the O_RDONLY or O_WRONLY flag set, the O_NDELAY flag results in the following actions:

- If O_NDELAY is set, an open for reading-only returns without delay. An open for writing-only returns an error if no process currently has the file open for reading.

- If O_NDELAY is clear, an open for reading-only blocks until a process opens the file for writing. An open for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line, the O_RDONLY flag results in the following actions:

- If O_NDELAY is set, the open returns without waiting for a carrier.
- If O_NDELAY is clear, the open blocks until a carrier is present.

O_NONBLOCK
: With pipes, functions like O_NDELAY, but eliminates the ambiguity of 0 bytes transferred, which means both end-of-file and no data available.

When the O_NDELAY flag is used while opening a pipe and a read from the pipe is performed, if the read returns a 0 value, the user's program cannot determine whether this situation means an end-of-file condition or whether there is no data currently in the pipe to read since a return value of 0 can indicate either condition.

When the O_NONBLOCK flag is used while opening a pipe and a read from the pipe is performed, if an end-of-file condition is encountered, the read returns a value of 0. If there is simply no data in the pipe to read, the read returns a value of −1.

When the O_NONBLOCK flag is used while opening a migrated file (file type IFOFL or IFOFD), the open does not block waiting for the file to be recalled from offline media. The open will return a value of −1 and set errno to EDMNBLK while the recall of the file from the media is in progress. Subsequent open calls with this flag will continue to return with this status until the file is fully recalled, at which point the open proceeds normally.

O_NOCTTY
: If used while opening a terminal device, open does not cause the terminal device to become the controlling terminal for the process.

O_BIG
: Allows a user to specify that a file is big when it is created, rather than wait for the file to grow large enough for the system to categorize it as big.

The BIGFILE parameter (defined in the header file sys/param.h) specifies the size in bytes past which a file is considered big.

File space, when allocated, comes from the secondary partition area (see mkfs(8)), if such an area is defined.

O_APPEND
: If set, the file pointer is set to the end of the file before each write. In addition, the file must be open for writing; see O_RDWR and O_WRONLY.

`O_CREAT` This option requires a third argument, *mode*, which is of type `mode_t`. If the file specified by *path* exists, this flag has no effect, except as noted under `O_EXCL` in the following. Otherwise, the file is created; the file's user ID is set to the effective user ID of the process, the file's group ID is set to the group ID of the directory in which the file is created, and the low-order 12 bits of the file mode are set to the value of *mode*, modified as follows (see `creat`(2)): all bits set in the process' file mode creation mask are cleared (see `umask`(2)).

`O_TRUNC` If the file exists, its length is truncated to 0, and the mode and owner are unchanged.

`O_EXCL` If `O_EXCL` and `O_CREAT` are set, `open` fails if the file exists.

`O_PLACE` If the file specified by *path* exists, this flag has no effect; otherwise, the `cbits` and `cblks` parameters, if passed, are used to establish file system partition residency and the number of blocks allocated in each partition.

Multiple set bits indicate that the file is to be striped on the specified partitions.

`O_RESTART` If `O_RESTART` and `O_CREAT` and the process has appropriate privilege, the file is created as a restart file (see `restart`(2) and `chkpnt`(2)).

`O_SSD` If `O_SSD` is set, all subsequent I/O is done from the users' secondary data segment (SDS) memory instead of main memory. The I/O is done with the backdoor channel on the IOS if it is configured, or else it uses the sidedoor (which is a backdoor software emulator) (Cray PVP systems except for CRAY EL series, CRAY J90 series, and CRAY T90 series).

Addresses passed on the I/O requests are interpreted as word addresses relative to the beginning of the process' SDS field length (see `ssbreak`(2)). The addresses must be on a 512-word boundary.

The count passed is still a byte count, but it must be a multiple of 4096 bytes.

This flag is not supported across NFS-mounted file systems. If the `O_SSD` flag is used to open a file accessed via NFS, the `EINVAL` error code is returned.

`O_SYNC` When a regular file is opened, this flag affects subsequent writes. If set, each `write`(2) waits for both the file data and file status to be physically updated.

`O_T3D` This flag controls memory usage when the high-speed (HISP) channel connects the I/O subsystem (IOS) and CRAY T3D system. If the `O_T3D` flag is set, all subsequent I/O is performed from the user's CRAY T3D memory instead of secondary data segments (SDS) or the main memory of the Cray host computer system. The I/O is done via backdoor channel on the IOS only. Sidedoor (which is a backdoor software emulator) is not supported.

Addresses passed on the I/O requests are interpreted as CRAY T3D memory addresses.

The count passed is still a byte count but must be a multiple of disk sector size of the referenced device.

`O_WELLFORMED` Used with `O_RAW` to force read and write requests that are not well-formed to fail. If ill-formed I/O requests are not specified with `O_RAW`, they are buffered without any notification to the user.

`O_SFS_DEFER_TM`

This flag is valid only for shared file systems (SFSs). It can reduce file system overhead by declaring to the UNICOS kernel that updates to file inode time stamps may be done less frequently than they otherwise would. If it is not critical to the application that the time stamps returned by the `stat` or `fstat` system calls be highly accurate, then setting this flag on very active files is advisable. The number of inode updates to media is reduced, which implies a reduction in overhead and a corresponding increase in performance.

`O_SFSXOP` This flag grants exclusive open lock on an SFS file system.
On return from the `open` the user process is guaranteed that no other process in the SFS cluster has this file open. While that process owns the open lock, the process may execute an `unlink`(2) system call on the file, thus causing all other pending `open` calls for this file to fail with an `ENOENT` error. If a process tries to open a file without the `O_SFSXOP` flag when the file is already open by another process with an exclusive open lock, the resulting behavior is determined by the presence or absence of either the `O_NDELAY` or the `O_NONBLOCK` flags. If either is set on the attempted open, the open will fail with `EAGAIN`. If neither flag is set, the process will sleep until the file has been closed by all other processes on all machines. If the same process opens a file with the exclusive open flag, and then attempts a subsequent non-exclusive open, the second open attempt will fail with `EDEADLK`.

mode Sets the bit pattern denoting the file's access permission. The following are valid patterns.

04000 Sets user ID on execution.

020#0 Sets group ID on execution if # is 7, 5, 3, or 1. Enables mandatory file or record locking if the file is an ordinary file and # is 6, 4, 2, or 0.

00400 Reads by owner.

00200	Writes by owner.
00100	Executes (or searches if a directory) by owner.
00070	Reads, writes, and executes (searches) by group.
00007	Reads, writes, and executes (searches) by others.

cbits Specifies the bits of the *cbits* argument correspond (starting with 2ˆ0 or the rightmost bit in the argument) in the file system. Multiple set bits indicate that the file is to be striped on the specified partitions. `O_PLACE` and `O_CREAT` must be set if *cbits* is passed as an argument to `open`( ).

If the file system uses primary and secondary partitions, you should not specify any *cbits* bits for the primary partitions. The file system uses the primary partitions to hold file system metadata (that is, inodes and directories) and the data for small files; it uses the secondary partitions to hold data for large files. If *cbits* bits are specified for primary partitions, those partitions may fill up and thus prevent the file system from creating any new files.

The primary partitions are always the first partitions in a file system. For example, if a file system has 2 primary partitions and 8 secondary partitions, partitions 0 and 1 are primary, and partitions 2 through 9 are secondary.

In a C program, you can use the `statfs`(2) system call to identify the primary and secondary partitions. The `df`(1) command with the `-p` option can be used to identify the primary and secondary partitions.

cblks Specifies the number of 512-word blocks allocated in each partition (as specified in *cbits* per stripe). This number is rounded up to the nearest multiple of the sector size. `O_PLACE` and `O_CREAT` must be set if *cblks* is passed as an argument to `open`().

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across `exec`(2) system calls. See `fcntl`(2).

No process may have more than `OPEN_MAX` file descriptors open simultaneously.

To ensure that data is written to disk immediately, either the `O_SYNC` flag or both the `O_RAW` and the `O_LDRAW` flags should be used.

NOTES

Only a process with appropriate privilege can create a restart file.

Only a process with appropriate privilege can open restricted devices.

If the file has an access control list (ACL), users who are not the file owner have permissions checked with respect to the access control list. Users who are not affected by entries in the access control list have permissions checked with respect to the other mode bits.

Permission to the file is also based on a comparison between the active security label of the process and the security label of the file. These comparisons are summarized as follows:

- For read or execute permission, the active security label of the process must dominate the security label of the file.
- For write permission, the active security label of the process must equal the security label of the file.
- For read permission to pipes, the active security label of the process must be equal to the security label of the file if the `SECURE_PIPE` configuration option is enabled; otherwise, the active security label of the process must dominate the security label of the file.

If the file is a labeled device, the active security label of the process must fall within the security label range of the device.

Only appropriately authorized users are granted permission to files that are in the OFF state or that are multilevel.

The process must be granted search permission to every component of the path prefix via the permission bits and access control list. The process must be granted search permission to every component of the path prefix via the security label.

If `FSETID_RESTRICT` is enabled, only processes with appropriate privileges can be granted write permission to set-user-ID or set-group-ID files.

To create a file, the calling process must have write permission to the file's parent directory via the permission bits and access control list. To create a file, the calling process must have write permission to the file's parent directory via the security label.

To create a file, the process must specify a file system that has been labeled as secure. The active security label of the process must fall with the security label range of the file system.

A process with the effective privileges shown are granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted read, execute, or write permission to the file via the permission bits and access control list.
`PRIV_FSETID`	If `FSETID_RESTRICT` is enabled, the process is granted write permission to the set-user-ID or set-group-ID file or is allowed to create a set-user-ID or set-group-ID file.
`PRIV_IO`	The process is allowed to open restricted devices.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_READ`	The process is granted read or execute permission to the file via the security label.

`PRIV_MAC_WRITE`	The process is granted write permission to the file via the security label. For file creation, the process is granted write permission to the file's parent directory via the security label.
`PRIV_RESTART`	The process is allowed to create a restart file.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted read, execute, or write permission to the file. The super user is allowed to create a restart file. The super user or a process with the `suidgid` permission can override the restriction enabled by the `FSETID_RESTRICT` system configuration option. The super user is allowed to open restricted devices.

RETURN VALUES

Upon successful completion of `open`, the file descriptor is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `open` system call fails to open the specified file if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`EACCES`	The *oflag* permission is denied for the specified file.
`EACCES`	The file is in the `OFF` state and the calling process does not have appropriate privilege.
`EACCES`	The file is a multilevel file and the calling process does not have appropriate privilege.
`EACCES`	The security label of the file does not allow the requested access.
`EACCES`	If the `FSETID_RESTRICT` and `PRIV_SU` configuration options are enabled, the process does not have appropriate privilege to gain write permission to the set-user-ID or set-group-ID file.
`EACCES`	The tape device file does not reside in directory `/dev/tape`, is a diagnostic device, or the device is not configured down.
`EAGAIN`	The file exists, mandatory file and record locking is set, and there are outstanding record locks on the file (see `chmod`(2)).
`EBUSY`	Device is already open.
`EDMNBLK`	`O_NONBLOCK` is set and the specified file is migrated offline.
`EDMOFF`	The specified file is offline, and the data migration facility is not configured in the system.
`EEXIST`	`O_CREAT` and `O_EXCL` are set, and the specified file exists.
`EFAULT`	The *path* argument points outside the allocated process address space.

`EFLNEQ`	The active security label of the process falls outside the security label range of the specified file system.
`EFSNOTEXCL`	The caller is attempting to open a file with `O_EXCL`, and other processes have the file open.
`EINTR`	A signal was caught during the `open` system call.
`EINTR`	The `open` system call was interrupted.
`EINVAL`	`O_APPEND` was specified in *oflag*, and the file to be opened is a process in the */proc* file system.
`EINVAL`	`O_SSD` was used to open a file accessed via NFS.
`EISDIR`	The specified file is a directory, and *oflag* is write or read and write.
`EMANDV`	The security label range of a device does not allow the requested access.
`EMFILE`	`OPEN_MAX` file descriptors are currently open.
`ENOENT`	`O_CREAT` is not set, and the specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`ENXIO`	The specified file is a character special or block special file, and the device associated with this special file does not exist.
`ENXIO`	`O_NDELAY` is set, the specified file is a FIFO special file, `O_WRONLY` is set, and no process has the file open for reading.
`EOFFLIN`	The specified file is offline, and automatic file retrieval is disabled.
`EOFLNDD`	The specified file is offline, and the data management daemon is not currently executing.
`EOFLNNR`	The specified file is offline, and it is currently unretrievable.
`EPERM`	`O_CREAT` and `O_RESTART` are set, and the effective user ID of the caller is not super user.
`EPERM`	The caller does not have appropriate privilege to open a restricted device.
`EQACT`	A file or inode quota limit was reached for the current account ID.
`EQGRP`	A file or inode quota limit was reached for the current group ID.
`EQUSR`	A file or inode quota limit was reached for the current user ID.
`EROFS`	The specified file resides on a read-only file system, and *oflag* is write or read and write.
`ESYSLV`	The specified file system has not been labeled as a secure file system.
`ETPDCNF`	The tape subsystem has not been configured.
`ETPD_BAD_REQT`	There is no path to the device.

`ETXTBSY` The text file is busy.

FORTRAN EXTENSIONS

The `open` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER oflag, mode, OPEN, I
I = OPEN (path, oflag, mode)
```

The *path* argument may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFOPEN`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

The following examples illustrate different uses of the `open` system call.

Example 1: This file, specified by its full path name, is opened for reading only and uses the default buffered I/O. That is, data passes through the system buffer cache because the `O_RAW` flag has not been specified.

```
int fd
fd = open("/usr/include/fcntl.h", O_RDONLY);
```

Example 2: This `open` request creates a file named `newfile` in the current directory (relative path name supplied). The file is given permissions of 0644 adjusted by the current user file-creation mode mask value. All write operations to the file are appended onto the end of the file.

If `newfile` already exists in the current directory, that file is opened for writing only.

```
int fd;
fd = open("newfile", O_WRONLY | O_CREAT | O_APPEND, 0644);
```

Example 3: The file, whose path name is found at the address specified by the pointer `fptr`, is created if it does not currently exist. If the file already exists, it is opened with its contents truncated.

All write operations to the file bypass the system buffer cache (raw mode) and write directly to the device.

```
int fd;
char *fptr;
fd = open(fptr, O_WRONLY | O_TRUNC | O_CREAT | O_RAW, 0640);
```

Example 4: This `open` request creates `newfile` for writing only.

The exclusive create feature (`O_EXCL`) indicates that if a file named `newfile` already exists in the current directory, the `open` request fails.

```
int fd;
fd = open("newfile", O_WRONLY | O_CREAT | O_EXCL, 0644);
```

Example 5: The following `open` request with the `O_SYNC` flag forces each succeeding write operation (for `datafile`, the file being opened) to wait until the output data has reached the physical device. By default, output data is staged in the system's buffer cache (buffered I/O) and does not reach the device immediately (called delayed I/O).

```
int fd;
fd = open("datafile", O_WRONLY | O_CREAT | O_SYNC, 0600);
```

Using the `O_SYNC` feature impairs I/O performance.

Example 6: The `O_SSD` flag on an `open` request enables data residing in a user's secondary data segments (SDS) area on the SDS to be written directly to the user's data file. For the following `open` request, succeeding write operations transfer data from the user's SDS area (rather than from the user's process memory) to the user's file `sdsdata`. The `write`(2) operation transfers 10 blocks (40,960 bytes) of data, starting at block position 1 (relative word location 512) of the user's SDS area, to the file `sdsdata`.

```
int fd;
fd = open("sdsdata", O_WRONLY | O_CREAT | O_SSD, 0644);
write(fd, 512, 40960);
```

SEE ALSO

`chkpnt`(2), `chmod`(2), `close`(2), `creat`(2), `dup`(2), `exec`(2), `fcntl`(2), `ialloc`(2), `lseek`(2), `mknod`(2), `read`(2), `restart`(2), `setdevs`(2), `ssbreak`(2), `statfs`(2), `umask`(2), `unlink`(2), `write`(2)

`umask`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`PXFOPEN`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

`mkfs`(8), `spdev`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`openi` – Opens a file by using the inode number

SYNOPSIS

`int openi (long` *dev*`, long` *ino*`, long` *gen*`, long` *uflag*`);`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `openi` system call presents the user with a flat view of all native UNICOS file systems currently mounted. Rather than use the directory tree structure to search through directories for a file, `openi` provides access by inode number.

The `openi` system call accepts the following arguments:

dev Specifies the device number as built by the `makedev` macro that is defined outside of the kernel.

ino Specifies an inode number for the file as reported by the `ls –i` command.

gen Specifies the generation number of the inode. This provides a unique identification for a specific file. The generation number changes when an inode is reused. To print the inode generation values, use the `fck`(1) command with the `-i` and `-l` options.

uflag Specifies the open flags. These are bit values of the form `O_`*name* that are defined in the `fcntl.h` file.

NOTES

Only a process with appropriate privilege can use this system call.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call.

A process with the `PRIV_MAC_READ` and `PRIV_DAC_OVERRIDE` effective privileges are allowed to use this system call. See the effective privilege discussion in the NOTES section of the `open`(2) man page for additional privilege requirements. The `open`(2) search access discussions do not apply to this system call.

RETURN VALUES

If `openi` completes successfully, a nonnegative integer is returned which may be used in further I/O operations. Otherwise, `openi` returns a negative value, and `errno` is set to indicate the error.

ERRORS

The `openi` system call fails to open the specified file if one of the error conditions listed on the `open`(2) man page occurs.

EXAMPLES

The following code fragment shows how `openi` is used in a program that examines restart files. The `rv_fs` field is a file system identifier. The `rv_fid` field contains an *ino gen* pair.

```
fd = openi( usr_makedev(krn_major(rvp->rv_fs.val[0]),
                        krn_minor(rvp->rv_fs.val[0])),
            rvp->rv_fid.fid_data[0],
            rvp->rv_fid.fid_data[1], 0);
```

FILES

`/usr/include/fcntl.h`	Contains symbol descriptions for the `open`(2) system call
`/usr/include/sys/sysmacros.h`	Contains a description of the `makedev` macro

SEE ALSO

`open`(2)

`fck`(1), `ls`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`fsck`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`pathconf`, `fpathconf` – Determines value of file or directory limit

SYNOPSIS

```
#include <unistd.h>
long pathconf (const char *path, int name);
long fpathconf (int fildes, int name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `pathconf` and `fpathconf` system calls provide a method for an application to determine the current value of a configurable limit or option (variable) that is associated with a file or directory.

The `pathconf` and `fpathconf` system calls accept the following arguments:

path Points to the path name of a file or directory.

name Represents the variable to be queried relative to that file or directory.

fildes Specifies an open file descriptor.

The values for *name* are listed below with a brief description of the value each returns.

`_PC_LINK_MAX` Returns the maximum number of links allowed per file.

`_PC_MAX_CANON` Returns the maximum number of bytes that can be read from a terminal device in canonical mode. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.

`_PC_MAX_INPUT` Returns the maximum number of bytes that can be read from a terminal device in raw mode. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.

`_PC_NAME_MAX` Returns the maximum number of characters in a file name relative to the file system containing the file specified as the first argument.

If *path* or *fildes* refers to a directory, the value returned applies to the file names within the directory. The behavior is undefined if *path* or *fildes* does not refer to a directory.

`_PC_PATH_MAX` — Returns the maximum number of characters in a path name relative to the file system containing the file specified as the first argument.

The behavior is undefined if *path* or *fildes* does not refer to a directory. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative path name when the specified directory is the working directory.

`_PC_PIPE_BUF` — Returns the maximum number of bytes that can be written to a pipe to be assured that the write is atomic. If the number of bytes written to a pipe is less than the value returned by the `_PC_PATH_MAX` request, then the writing process is assured that the data written is not interleaved with data from another process' write to the same pipe.

If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the referenced object itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildes* refer to any other type of file, the behavior is undefined.

`_PC_CHOWN_RESTRICTED` — Returns value 1 if `chown`() is a restricted operation; returns −1 if `chown`() is unrestricted. The system can be configured such that only users granted permission may use `chown`().

If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

`_PC_NO_TRUNC` — Returns value 1 if path name components longer than `NAME_MAX` generate an error. Returns value −1 if path name components longer than `NAME_MAX` do not generate errors (that is, path name components longer than `NAME_MAX` are truncated to `NAME_MAX` characters without causing an error condition).

If *path* or *fildes* refers to a directory, the value returned applies to the file names within the directory. The behavior is undefined if *path* or *fildes* does not refer to a directory.

`_PC_VDISABLE` — Returns the disable character for terminal devices. Terminal special control characters defined in `termios.h` can be disabled using this character value. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.

NOTES

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

To be granted search permission to a component of the path prefix (for the `pathconf` system call), the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list. (Only for `pathconf` system call.)
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label. (Only for `pathconf` system call.)
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call and is granted search permission to every component of the path prefix. If the `PRIV_SU` configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If *name* is an invalid value, the `pathconf` and `fpathconf` system calls return a value of –1.

If the variable corresponding to *name* has no limit for the path or file descriptor, `pathconf` and `fpathconf` return a value of –1 without changing `errno`.

If *path* determines the value of *name* and *name* is not associated with the file specified by *path*, or if the process did not have the appropriate privileges to query the file specified by *path*, or if *path* does not exist, `pathconf` returns a value of –1.

If *fildes* determines the value of *name* and *name* is not associated with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, `fpathconf` returns a value of –1. Otherwise, `pathconf` and `fpathconf` return the current variable value for the file or directory without changing `errno`.

ERRORS

The `pathconf` or `fpathconf` system call returns a value of –1 and sets `errno` to the corresponding value if one of the following conditions occurs:

Error Code	Description
`EINVAL`	The value of the name is not valid.
`EPERM`	The process does not have appropriate privilege to use this system call.

The `pathconf` system call returns a value of −1 and sets `errno` to the corresponding value if one of the following conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
EACCES	The process is denied read permission to the file via the security label.
EINVAL	The variable name is not associated with the specified file.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or a path name component is longer that NAME_MAX when _POSIX_NO_TRUNC is in effect.
ENOENT	The specified file does not exist or the *path* argument points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.

The `fpathconf` system call returns a value of −1 and sets `errno` to the corresponding value if one of the following conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not a valid file descriptor.
EBADF	The process is denied read permission to the file via the security label.
EINVAL	The variable name is not associated with the specified file.

EXAMPLES

This example illustrates various applications of the `pathconf` system call for files and directories residing in the user's home (HOME) directory.

```
#include <unistd.h>

main()
{
    char path[100], *ptr;
    long name_max;

    if (ptr = getenv("HOME")) {
         strcpy(path, ptr);
    }
    else {
         fprintf(stderr, "getenv failed to locate HOME!\n");
         exit(1);
    }

    printf("Configurable parameters for files within %s:\n", path);
    printf("    maximum # of links = %ld\n", pathconf(path, _PC_LINK_MAX));
    name_max = pathconf(path, _PC_NAME_MAX);
    printf("    maximum # of chars in a filename = %ld\n", name_max);
    printf("    maximum # of chars in a path name = %ld\n",
                                                  pathconf(path, _PC_PATH_MAX));
    printf("    maximum # of bytes for atomic writes to a pipe = %ld\n",
                                                  pathconf(path, _PC_PIPE_BUF));
    if (pathconf(path, _PC_CHOWN_RESTRICTED) == -1) {
         printf("    chown is unrestricted\n");
    }
    else {
         printf("    chown is restricted\n");
    }
    if (pathconf(path, _PC_NO_TRUNC) == -1) {
         printf("    path name components longer than %d char’s ", name_max);
         printf("do not generate errors;\n");
         printf("        (that is, the system will use only the first ");
         printf("%d characters)\n\n", name_max);
    }
    else {
         printf("    path name components longer than %d char’s ", name_max);
         printf("generate errors\n\n");
    }

    printf("Configurable parameters for terminals:\n");
    printf("    maximum # bytes for canonial reads = %ld\n", fpathconf(0, _PC_M AX_CANON));
    printf("    maximum # bytes for raw reads = %ld\n", fpathconf(0, _PC_MAX_IN PUT));
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `pathconf` and `fpathconf` system calls

SEE ALSO

`sysconf`(2)

`getconf`(1), `sysconf`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`pause` – Suspends process until signal

SYNOPSIS

```
#include <unistd.h>
int pause (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `pause` system call suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process. The `pause` system call causes an implicit `sigon` (see `sigoff`(3C)).

On Cray MPP systems, the `pause` system call suspends the process only for the PE on which it is called. It has no effect on any other PE of the application.

RETURN VALUES

If the signal causes termination of the calling process, `pause` does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function (see `signal`(2)), the calling process resumes execution from the point of suspension, a value of –1 is returned, and `errno` is set to `EINTR`.

FORTRAN EXTENSIONS

The `pause` system call can be called from Fortran as a function:

```
INTEGER PAUSE, I
I = PAUSE ( )
```

Alternatively, `pause` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable:

```
CALL PAUSE ( )
```

EXAMPLES

This example shows how to use the `pause` system call to make a process wait for a specific signal.

The `pause` system call suspends a process until any signal is received.

In this example, the use of the `sigsetmask`(2) system call with `pause` enables the request to delay the program until receipt of a `SIGUSR1` signal.

```
#include <signal.h>
#include <unistd.h>

int omask, nmask;

main()
{
     void catch(int signo);

     signal(SIGUSR1, catch);

     /* Process performs work here, but after finishing work and before
        proceeding, it needs to wait for a SIGUSR1 signal to be sent
        from another process. */

     nmask = ~sigmask(SIGUSR1);/* enable all bits in mask except SIGUSR1 */
     omask = sigsetmask(nmask);/* hold all signals except SIGUSR1 */
     pause();                  /* wait for SIGUSR1 signal only */

     /* Work continues here after waiting and catching SIGUSR1 signal */
}

void catch(int signo)
{
     sigsetmask(omask);        /* restore signal hold mask after signal
                                  is received */
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `pause` system call

SEE ALSO

`alarm`(2), `kill`(2), `signal`(2), `sigsetmask`(2), `wait`(2)

`sigoff`(3C) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

pipe – Creates an interprocess channel

SYNOPSIS

```
#include <unistd.h>
int pipe (int fildes[2]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The pipe system call creates an I/O mechanism called a pipe and returns two file descriptors. It accepts the following argument:

fildes Specifies the file descriptors returned. These are *fildes*[0] and *fildes*[1]; *fildes*[0] is opened for reading, and *fildes*[1] is opened for writing.

A maximum of MAXPIPE data is buffered by the pipe before the writing process is blocked. MAXPIPE, defined in the /usr/src/uts/c1/cf/config.h header file, specifies the number of data blocks (4096 bytes each) reserved in the pipe's kernel buffer space. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a FIFO special file basis.

NOTES

The active security label of the process must fall within the security label range of the root file system.

RETURN VALUES

If pipe completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The pipe system call fails if one of the following error conditions occurs:

Error Code	**Description**
EMFILE	OPEN_MAX –1 or more file descriptors are currently open.
ENFILE	The system file table is full.
ENOSPC	During a write(2) to an ordinary file, the free space left on the device was exhausted.

`EFLNEQ` The active security label of the process does not fall within the security label range of the root file system.

FORTRAN EXTENSIONS

The `pipe` system call can be called from Fortran as a function:

```
INTEGER fildes(2), PIPE, I
I = PIPE (fildes)
```

EXAMPLES

This example shows how to use the `pipe` system call to create a system pipe. (Some system calls in the example are not supported on Cray MPP systems.) Because a system pipe can transfer data only between related processes, such as a parent and child or siblings, this example shows the essential elements of parent and child processes needed for data transfer by a system pipe.

```
/* This is the parent side of the system pipe example. It delivers data to
   the child process using a system pipe. */
#include <stdio.h>
#include <unistd.h>

main()
{
     int fd[2];
     char rfd[10], wfd[10];

     if (pipe(fd) == -1) {            /* create system (unnamed) pipe */
          perror("creating system pipe failed");
          exit(1);
     }

     if (fork() == 0) {               /* create child process */
          sprintf(rfd, "%d", fd[0]); /* convert pipe's file descriptors - */
          sprintf(wfd, "%d", fd[1]); /* to strings to pass as arguments */
          execl("child_prog", "child_prog", rfd, wfd, 0);
          perror("execl failed");
          exit(1);
     }

     close(fd[0]);                    /* parent closes its read access to
                                         pipe since the  parent will write
                                         to the pipe */

     /* In this part of program, the parent writes to fd[1] to deliver data
         to the pipe. */

     close(fd[1]);                    /* parent closes its write access to pipe -
```

```
                                         required for the child to detect an EOF
                                         condition */
     wait((int *)0);                  /* parent waits for child to complete */
}

/* This is the child side of the system pipe example. It receives data
   from the parent process on a system pipe. */


main(int argc, char *argv[])
{
     int rfd, wfd;

     rfd = atoi(argv[1]);             /* since pipe's file descriptors were
     wfd = atoi(argv[2]);             /* passed as arguments, the string
                                         arguments are converted back to
                                         integers */

     close(wfd);                      /* child closes its write access
                                         to pipe since the child will read
                                         from the pipe - also required for child
                                         to detect an EOF condition */

     /* In this part of program, the child reads from rfd (fd[0]) to receive
         data from the pipe. */

     close(rfd);                      /* child closes its read access
                                         to the pipe and then exits */
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `pipe` system call
`/usr/src/uts/c1/cf/config.h`	Contains `MAXPIPE` definition

SEE ALSO

`read`(2), `write`(2)

`ksh`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`plock` – Locks process in memory

SYNOPSIS

```
#include <sys/lock.h>
int plock (int op);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `plock` system call allows the calling process to lock itself in memory. Locked processes are immune to all routine swapping. The `plock` system call also allows these segments to be unlocked. Only a process with appropriate privilege can use this system call.

The `plock` system call accepts the following argument:

op Specifies a locking option. The following are valid *op* values:

`DATLOCK`	Locks data in memory (data lock).
`DLYSHUFFLE`	Locks process in memory, movable (process lock); does not force process to low-memory address immediately.
`NOSHUFFLE`	Locks process in memory, not movable (process lock).
`PROCLOCK`	Locks process in memory, movable (process lock).
`TXTLOCK`	Locks text in memory (text lock).
`UNLOCK`	Removes locks.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_RESOURCE`	The process is allowed to use this system call.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_PLOCK` permbit is allowed to use this system call.

RETURN VALUES

If `plock` completes successfully, a value of 0 is returned to the calling process; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `plock` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EINVAL`	The *op* argument is equal to `PROCLOCK`, and a process lock already exists on the calling process.
`EINVAL`	The *op* argument is equal to `NOSHUFFLE`, and a process lock already exists on the calling process.
`EINVAL`	The *op* argument is equal to `TXTLOCK`, and a text lock or a process lock already exists on the calling process.
`EINVAL`	The *op* argument is equal to `DATLOCK`, and a data lock or a process lock already exists on the calling process.
`EINVAL`	The *op* argument is equal to `UNLOCK`, and no type of lock exists on the calling process.
`EINVAL`	The *op* argument is equal to `TXTLOCK` or `DATLOCK`, and the program is not compiled with split code and data.
`EINVAL`	The *op* argument is equal to `DLYSHUFFLE`, and a process lock already exists on the calling process.
`EPERM`	The process does not have appropriate privilege to use this system call.

FORTRAN EXTENSIONS

The `plock` system call can be called from Fortran as a function:

```
INTEGER op, PLOCK, I
I = PLOCK (op)
```

SEE ALSO

`exec`(2), `exit`(2), `fork`(2)

NAME

policy – Returns or sets information on the CPU allocation policy

SYNOPSIS

```
#include <sys/types.h>
#include <sys/share.h>

int policy (int function, void *address, int action);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The policy system call allows site selection of CPU allocation policy (deferred implementation) without requiring changes to the priority adjustment mechanism in the UNICOS kernel. This system call is also used by shrdaemon(8) to access the sh_consts structure in the kernel.

The policy system call accepts the following arguments:

function Specifies the CPU allocation policy to be affected by the *action*. The following policies are available:

- FAIR_SHARE Specifies the standard (default) CPU allocation policy for the fair-share scheduler.
- BANK_POINTS (Deferred implementation) Specifies the *bank points* CPU allocation policy, which provides a "bank account" of resources that is depleted through use.

address Specifies the location of the policy-definition structure. For example, the sh_consts structure, defined in the include file sys/share.h, is used for the standard fair-share information.

action Specifies the action to be performed. The following values are available:

- GET_COSTS Retrieves the contents of the system sh_consts structure.
- SET_COSTS Sets the contents of the system sh_consts table. This action acts on only those parameters that can be changed at the user level; for example, the counts and maximum value fields are not updated. Only a process with appropriate privilege can specify this function.
- MOD_MXUSG Sets the maximum usage value field (sc_mxcusage) of the shconst structure. Only a process with appropriate privilege can specify this function. This action is not valid with the BANK_POINTS function.

NOTES

The implementation of the BANK_POINTS function is deferred. The following actions are currently available (the example address &shconsts assumes a declaration of the form struct sh_consts shconsts):

- policy(FAIR_SHARE, &shconsts, GET_COSTS)
- policy(FAIR_SHARE, &shconsts, SET_COSTS)
- policy(FAIR_SHARE, &shconsts, MOD_MXUSG)

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_RESOURCE	The process is allowed to specify the SET_COSTS and MOD_MXUSG functions.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_SYSPARAM permbit is allowed to specify the SET_COSTS and MOD_MXUSG functions.

RETURN VALUES

If policy completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The policy system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The *address* of the structure points to invalid data.
EINVAL	Either the *function* or *action* argument is invalid. This error is returned if the deferred BANK_POINTS function is specified.
EPERM	The process does not have appropriate privileges for the SET_COSTS and MOD_MXUSG actions.

SEE ALSO

share(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

shradmin(8), shrdaemon(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

`profil` – Generates an execution time profile

SYNOPSIS

```
#include <unistd.h>
void profil (long *buf, int bufsiz, int offset, int scale, int rate);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `profil` system call generates an execution time profile of the user's program. It accepts the following arguments:

buf Points to a memory area.

bufsiz Specifies the length (in bytes) of the memory area.

offset Specifies the number subtracted from the user's program counter (PC) every interval specified by *rate*.

scale Specifies the multiplier of the difference resulting from subtracting *offset* from the user's PC. It is interpreted as an unsigned, fixed-point fraction with binary point at the left. If the resulting number corresponds to a word inside *buf*, that word is incremented. The octal number 037777777777 gives a one-to-one mapping of PCs to words in *buf*; the octal number 017777777777 maps each pair of instruction words together; the octal number 02 maps all instructions onto the first word in *buf* producing a noninterrupting core clock.

rate Specifies the rate in microseconds at which the program is sampled. Default is 1/100 second if *rate* is 0 or is not specified (Cray PVP systems).

The `profil` system call is inoperative under the following conditions:

- An update in *buf* would cause a memory fault.
- The *bufsiz* argument is 0.
- The *scale* argument is 0 or 1.
- The `exec`(2) system call is executed.

The `profil` system call remains operative in both a child process and parent process after processing a `fork`(2) system call.

RETURN VALUES

None

FILES

`/usr/include/unistd.h`	Contains C prototype for the `profil` system call

SEE ALSO

`exec`(2), `fork`(2)

NAME

`ptrace` – Traces processes

SYNOPSIS

```
#include <unistd.h>
long ptrace (int request, int pid, long addr, long data);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `ptrace` system call provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging (see `adb`(1)). The child process behaves normally until it encounters a signal (see `signal`(2) for the list), at which time it enters a stopped state and its parent process is notified through `wait`(2). When the child process is in the stopped state, its parent process can examine and modify its core image, using `ptrace`. Also, the parent process can cause the child process either to terminate or to continue, with the possibility of ignoring the signal that caused it to stop.

The `ptrace` arguments are as follows:

request Identifies the action to be taken by `ptrace`. The following are valid values for *request*:

`0` This request must be issued by the child process if it is to be traced by its parent process. It turns on the child process' trace flag, stipulating that the child process should be left in a stopped state on receipt of a signal rather than the state specified by *func*; see `signal`(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. If the parent process does not expect to trace the child process, peculiar results occur.

The remainder of the requests can be used only by the parent process. The child process must be in a stopped state before these requests are made.

`1, 2` With these requests, the word at location *addr* in the address space of the child process is returned to the parent process. Requests 1 and 2 produce equal results. The *data* argument is ignored.

`3` With this request, the word at location *addr* in the child process' USER area in the system's address space (see the `sys/user.h` file) is returned to the parent process. Addresses in this area range from 0 to *sizeof* (structure user). The *data* argument is ignored. If *addr* is outside the USER area, this request fails.

`4, 5` With these requests, the value given by the *data* argument is written into the address space of the child process at location *addr*. Using either request 4 or 5 causes equal results. On successful completion, the value written into the address space of the child process is returned to the parent process.

`6` With this request, you can write a set of limited fields in the child process' USER area. The *data* argument gives the value to be written, and *addr* is the location of the entry. You can write the following entries:

- The general registers (that is, A, S, and V registers)
- The vector mask (VM) and vector length (VL) special registers
- B and T registers

`7` This request causes the child process to resume execution. If the *data* argument is 0, all pending signals, including the one that caused the child process to stop, are canceled before it resumes execution. If the *data* argument is a valid signal number, the child process resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be 1. If the *data* argument is 0, the child process resumes execution where it entered the stopped state. On successful completion, the value of *data* is returned to the parent process.

`8` This request causes the child process to terminate with the same consequences as those of `exit`(2).

`10` With this request, the word at location *addr* in the child process' UCOMM area in the system's address space (see `sys/user.h`) is returned to the parent process. Addresses in this area range from 0 to *sizeof* (structure `ucomm`). The *data* argument is ignored. If *addr* is outside the UCOMM area, this request fails.

`11` With this request, you can write a few entries in the child process' UCOMM area. The *data* argument gives the value to be written, and *addr* is the location of the entry. You can write the following entries:

- The semaphores
- Shared B registers
- Shared T registers

pid Specifies the process ID of the child process when *request* is 1 through 8. The *pid* argument is ignored when *request* equals 0.

addr Specifies a location that varies in meaning depending on the value of *request*.

data Specifies information supplied to or received from the target process; this information varies in meaning depending on the value of *request*.

To prevent fraud, `ptrace` inhibits the set user ID facility on subsequent `exec`(2) calls. If a traced process calls `exec`(2), it stops before executing the first instruction of the new image showing the `SIGTRAP` signal.

NOTES

On a UNICOS system using privilege assignment lists (PALs), a privileged process should not have itself traced. The security policy could be circumvented if a privileged process is traced by a nonprivileged parent process. This is also true if the `PRIV_SU` configuration option is enabled, although the phrase "privileged process" means a process owned by `root` on a `PRIV_SU` system.

To retrieve information about a child process, the active security label of the process must be greater than or equal to the security label of the child.

To set information about or modify the state of a child process, the active security label of the process must equal the security label of the child.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_READ`	The process is allowed to retrieve information about a child process regardless of the security label of the child.
`PRIV_MAC_WRITE`	The process is allowed to set information about or modify the state of a child process regardless of the security label of the child.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override the security label restrictions.

CAUTIONS

Requests to read can return a valid data value of –1, which can be confused with an error return value. `errno` is cleared by the library interface; therefore, if `ptrace` returns –1 and `errno` is nonzero, an error has occurred.

RETURN VALUES

If `ptrace` completes successfully, any requested value is returned; otherwise, a value of –1 is returned to the parent process, and the `errno` of the parent process is set to indicate the error.

ERRORS

The `ptrace` system call fails if one of the following conditions occurs:

Error Code	**Description**
`EIO`	The *request* argument is an illegal number, or when *request* is 7, *data* is not 0 or a valid signal number.
`ESRCH`	The *pid* argument identifies a child process that does not exist or that has not executed a `ptrace` with request 0.
`ESRCH`	The process does not meet security label requirements and does not have appropriate privilege.

FORTRAN EXTENSIONS

The `ptrace` system call can be called from Fortran as a function:

```
INTEGER request, pid, addr, data, PTRACE, I
I = PTRACE (request, pid, addr, data)
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `ptrace` system call

SEE ALSO

`exec`(2), `exit`(2), `signal`(2), `wait`(2)

`adb`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`proc`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`ptyrecon` – Manages pty reconnection

SYNOPSIS

```
#include <sys/ptyrecon.h>
int ptyrecon (int cmd, struct reconcntl *reconp);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `ptyrecon` system call enables or disables pty disconnection, and reconnects, searches, or hangs up disconnected sessions.

If disconnection is enabled, a session does not disappear on a pty master side close operation, but remains disconnected for a specified amount of time. Users can later search, reconnect to, or hang up disconnected sessions.

The `ptyrecon` system call accepts the following arguments:

cmd Indicates the operation to be performed.

`RECON_ENABLE` Enables reconnection on a pty. After closing the connection (by using the `telnet close` command or equivalent) the terminal remains in a disconnected state for the time indicated in the *distimeo* field of the `reconcntl` structure. If this value is 0, the default `DISTIMEO` is used.

`RECON_DISABLE` Disables reconnection that was enabled with `RECON_ENABLE`.

`RECON_HANGUP` Hangs up a terminal that is in a disconnected state.

`RECON_SEARCH` Fills the `reconcntl` structure with the data corresponding to the first disconnected pty greater than or equal to the pty number that was passed to the kernel. If the `reconcntl` flag `RECON_ANYUSER` is set, it returns the following message:

```
Disconnected session owned by any user (superuser only).
```

Otherwise, it returns the following message:

```
Sessions owned by the caller
```

`RECON_CONNECT` Connects the current terminal to a disconnected session in another pty, and terminates the current session.

reconp Points to a two-way communication structure. All of the operations act on the pty specified in the *pty* field of the `reconcntl` structure. If this field is set to `RECON_CURRPTY`, the operation acts on the current (controlling) pty. Following is the `reconcntl` structure, followed by a list of the possible operations:

```
struct reconcntl {
        int     pty;            /* pty number */
        int     uid;            /* session owner uid */
        pid_t   pid;            /* process-id of session leader */
        time_t  distime;        /* seconds disconnected */
        time_t  distimeo;       /* max disconnect time (sec) */
        int     flags;
};
```

RETURN VALUES

If `ptyrecon` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `ptyrecon` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EFAULT`	The *reconp* parameter is bad.
`EINVAL`	The *cmd* parameter is bad.
`ENOENT`	No more search entries exist.
`ENOTTY`	No controlling tty exists.
`ENXIO`	The pty number is bad.
`EPERM`	You are not allowed to perform the operation.

SEE ALSO

`ptyrecon`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`quotactl` – Manipulates file system quotas

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/quota.h>

int quotactl (char *spec, int request, char *arg);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `quotactl` system call manipulates disk quotas. The arguments are as follows:

spec Points to the name of the file system. The pointer may be either the device node name or the directory on which the device is mounted.

request Specifies type of request. Types are defined in **Request Types**.

arg Specifies address of a request-specific data structure described with each *request*. All structures are defined in the `sys/quota.h` file.

You can look at your own quota data and header information; however, only an appropriately privileged process can view all the records and change information. The requests that get and set information can be used only on file systems that have been mounted and activated with one of the `Q_ON_`*xxxx* requests. The following are valid values for *request*.

Request Types

The following types of requests are supported:

`Q_ON_COUNT` Appropriately privileged process only. Turns on quotas for the file system specified in *spec* but maintains only counts. *arg* points to the full name of the quota file to associate with this file system.

`Q_ON_INFORM` Appropriately privileged process only. Turns on quotas for the file system specified in *spec*, maintain counts, and issues warning and quota limit messages, but does not enforce quota limits. *arg* points to the full name of the quota file to associate with this file system.

`Q_ON_ENFORCE` Appropriately privileged process only. Turns on quotas for the file system specified in *spec*, maintains counts, issues warning and quota limit messages, and enforces quota limits. This is considered the normal mode of quota operation; the other modes are for evaluation or test use. *arg* points to the full name of the quota file to associate with this file system.

`Q_GETQUOTA` Returns the quota information for the ID specified in the `qf_entry.id` field from the file system specified in *spec*. If the request is not from an appropriately privileged process, only information related to the caller's *uid*, authorized *gid*s, and the currently active *acid*s can be obtained. *qf_magic* must be set to `QF_MAGIC` as defined in the `sys/quota.h` file. *arg* points to a `q_request` structure.

`Q_SETQUOTA` Appropriately privileged process only. Changes selected quota information for the ID specified in the `qf_entry.id` field on the file system specified in *spec*. Selection is through the `acct`, `group`, and `user` flag fields in the `q_request` structure. *qf_magic* must be set to `QF_MAGIC` as defined in the `sys/quota.h` file. *arg* points to a `q_request` structure.

For the `Q_GETQUOTA` and `Q_SETQUOTA` requests, *arg* points to a `q_request` structure defined in `sys/quota.h`:

```
struct q_request   {
    long     qf_magic;   magic number
    int      acct;       QFL_xxx account flags
    int      group;      QFL_xxx group flags
    int      user;       QFL_xxx user flags
    struct   qf_entry
             qf_entry;   quota record
};
```

The fields of `struct qf_entry` that contain valid information are indicated through the `acct`, `group`, and `user` fields. Only the values in the flagged fields are defined. The caller must set the correct `acct`, `group`, and `user` flags when making a `Q_SETQUOTA` request; the kernel sets the flags when returning information from a `Q_GETQUOTA` request. The following table shows the values that are returned and their associated meanings:

`QFL_F1`	(1<< 0)	Evaluator's field 1
`QFL_F2`	(1<< 1)	Evaluator's field 2
`QFL_F3`	(1<< 2)	Evaluator's field 3
`QFL_F4`	(1<< 3)	Evaluator's field 4
`QFL_F5`	(1<< 4)	Evaluator's field 5
`QFL_FQ`	(1<< 5)	File quota
`QFL_FR`	(1<< 6)	Soft file quota (runquota)
`QFL_FT`	(1<< 7)	File warning time
`QFL_FU`	(1<< 8)	File usage
`QFL_FW`	(1<< 9)	File warning
`QFL_IQ`	(1<< 10)	Inode quota
`QFL_IU`	(1<< 11)	Inode usage
`QFL_IW`	(1<< 12)	Inode warning

`Q_GETHEADER` Returns the header information for the file system specified in the *spec* argument. *qf_magic* must be set to `QF_MAGIC`. *arg* points to a `qf_header` structure.

`Q_SETHEADER` Appropriately privileged process only. Changes the header information for the file system specified in *spec*. All the header information, except for the `qf_name` field (which cannot be altered through this interface), is changed by this request; therefore, it is recommended that you use a `Q_GETHEADER` request to preset the `qf_header` structure to preserve information you do not want to change. *arg* points to a `qf_header` structure.

For the `Q_GETHEADER` and `Q_SETHEADER` requests, *arg* points to the following structure:

```
struct   qf_header  {
    long   qf_magic;            quota version identification
    struct q_header
           acct_h,              account header
           group_h,             group header
           user_h;              user header
    time_t qf_min_dm;           minimum data migration threshold
    uint   qflvl : 8,           Q_ON_DEFAULT enable level
           qf_eval : 8,         evaluator selector
           qf_spare : 48;       reserved
    long   hef1,                field 1 reserved for evaluator’s use
           hef2;                field 2 reserved for evaluator’s use
    uint   qf_qfnamesize;       size of qf_name[]
    uint   qf_hashents;         length of the hash table in entries
    off_t  qf_hashtaboffs;      offset of the hash table
    char   qf_name[PATH_MAX+1]; name of the quota file
};
```

The header contains all of the default values used for the quota control file used by the specified file system.

See the `sys/quota.h` file for a description of the fields.

`Q_FSINFO` Returns file system specific information. File size, quota enforcement level, and other information specific to a one quota control file is returned. *arg* points to a `q_fsinfo` structure.

```
struct  q_fsinfo  {
    int    count;    count of file systems using this quota file
    in     errors;   count of errors on this quota file
    long   level;    quota enforcement level
    long   size;     size of the quota file in bytes
};
```

`Q_GENINFO` Returns generic quota enforcement information in a `q_geninfo` structure pointed to by *arg*.

```
struct q_geninfo   {
    long   q_types;     mask of configured quota types
    long   q_nquota;    total number of quota entries
    long   q_curact;    current number of active quota entries
    long   q_maxact;    maximum number of active quota entries
    long   q_put;       qput calls
    long   q_updat;     qupdat calls
    long   q_get;       qget calls
    long   q_read;      qread calls
    long   q_cache;     count of inode cache flushes
    long   q_wait;      count of number of quota_wait sleeps
    long   q_readch;    count of hash chain reads
};
```

The configured quota types mask (`q_types`) has a value for each configured combination of ID classes, as shown in the following list:

Value	ID classes
`1`	User ID
`2`	Group ID
`3`	User ID and group ID
`4`	Account ID
`5`	User ID and account ID
`6`	Group ID and account ID
`7`	User ID, group ID, and account ID

`Q_ON_DEFAULT` Appropriately privileged process only. Turns on quotas for the file system specified in *spec*. The enforcement mode is the mode stored in the `qf_lvl` field of `struct qf_header` of the specified quota file. *arg* points to the name of the quota file to associate with this file system.

NOTES

To be granted read permission to the quota file, the caller's active security label must be greater than or equal to the security label of the quota file.

To be granted write permission to the quota file, the caller's active security label must equal the security label of the quota file.

To be granted `searcdh` permission to a component of the *spec* path prefix, the caller's active security label must be greater than or equal to the security label of the component.

A process with the effective privilege shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the *spec* path prefix via the permission bits and access control list.

`PRIV_DAC_OVERRIDE`	The process is granted read and write permission to the quota file via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to every component of the *spec* path prefix via the security label.
	The process is granted read permission to the quota file via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the quota file via the security label.
`PRIV_RESOURCE`	The process is allowed to specify `quotactl` requests that are restricted to processes with appropriate privilege.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override all file protections and is allowed to specify `quotactl` requests that are restricted to processes with appropriate privilege.

RETURN VALUES

If `quotactl` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `quotactl` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	The caller does not have read and/or write permission to the quota file.
`EACCES`	Search permission is denied for a component of the *spec* path prefix.
`EACCES`	The magic number in the quota file does not match the one used by the kernel.
`EBUSY`	The quota feature software is already running on file system *spec*.
`EFAULT`	The *arg* argument points outside the allocated process address space.
`EINVAL`	The *request* argument specified was not valid.
`EIO`	The kernel could not read and/or update the quota entry specified by argument *arg*.
`ENODEV`	The *spec* argument is not the root of a file system.
`ENOENT`	The quota feature software is not running on file system *spec*.
`EPERM`	The process does not have appropriate privilege to perform the requested action.
`EPERM`	The *spec* argument specifies a nonnative file system.

EXAMPLES

The following program illustrates two features of the `quotactl` system call: `Q_GETQUOTA` and `Q_GETHEADER`. The program is invoked specifying one argument, the path name of the file system for which quota information is to be retrieved.

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/quota.h>
#include <time.h>
#include <stdio.h>

main(int argc, char *argv[])
{
     struct q_request request;
     struct qf_header header;
     int id;

     if (argc < 2) {
          fprintf(stderr, "File system path name not supplied as argument\n");
          exit(0);
     }

     id = getuid();
     request.qf_entry.id = id;
     request.qf_magic = QF_MAGIC;

     if (quotactl(argv[1], Q_GETQUOTA, &request) == -1) {
          perror("quotactl (Q_GETQUOTA) failed");
          exit(1);
     }
     printf("file system name = %s\n", argv[1]);
     printf("account flags = %o\n", request.acct);
     printf("group flags = %o\n", request.group);
     printf("user flags = %o\n\n", request.user);
     if (request.user) {
          printf("Quotas for user %d are as follows:\n\n", id);
          if (request.qf_entry.user_q.f_quota == QFV_DEFAULT) {
               printf("The default file quota is in effect - see below\n");
          }
          else {
               printf("File quota = %ld blocks\n",
                                                request.qf_entry.user_q.f_quota);
          }
          printf("File usage = %ld blocks\n", request.qf_entry.user_q.f_use);
          if (request.qf_entry.user_q.f_warn == QFV_DEFAULT) {
               printf("The default file quota warning window is in effect ");
               printf("- see below\n");
          }
          else {
               printf("File quota warning window = %d blocks\n",
                                                 request.qf_entry.user_q.f_warn);
          }
          if (request.qf_entry.user_q.i_quota == QFV_DEFAULT) {
```

```
                printf("The default inode quota is in effect - see below\n");
            }
            else {
                printf("Inode quota = %d\n", request.qf_entry.user_q.i_quota);
            }
            printf("Inode usage = %d\n", request.qf_entry.user_q.i_use);
            if (request.qf_entry.user_q.i_warn == QFV_DEFAULT) {
                printf("The default inode quota warning window is in effect ");
                printf("- see below\n");
            }
            else {
                printf("Inode quota warning window = %d\n",
                                                  request.qf_entry.user_q.i_warn);
            }
            if (request.qf_entry.user_q.f_wtime != 0) {
                printf("Time when file warning was reached = %s\n",
                                         ctime(&request.qf_entry.user_q.f_wtime));
            }
        }

        header.qf_magic = QF_MAGIC;
        if (quotactl(argv[1], Q_GETHEADER, &header) == -1) {
            perror("quotactl (Q_GETHEADER) failed");
            exit(2);
        }
        printf("\nDefault quotas for file system %s:\n\n", argv[1]);
        printf("File quota = %ld blocks\n", header.user_h.def_fq);
        printf("File warning window = %d blocks\n", header.user_h.warn_fq);
        printf("File quota warning fraction = %3.1f\n", header.user_h.wf_fq);
        printf("Inode quota = %d\n", header.user_h.def_iq);
        printf("Inode warning window = %d\n", header.user_h.warn_iq);
        printf("Inode quota warning fraction = %3.1f\n", header.user_h.wf_iq);
    }
```

SEE ALSO

mount(2)

NAME

`read` – Reads from file

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

ssize_t read (int fildes, void *buf, size_t nbyte);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `read` system call tries to read a specified number of bytes from a file into a specified buffer. It accepts the following arguments:

fildes Specifies a file descriptor. It is obtained from an `accept`(2), `creat`(2), `dup`(2), `fcntl`(2), `open`(2), `pipe`(2), `socket`(2), or `socketpair`(2) system call

buf Points to the buffer into which the data is to be read.

nbyte Specifies the number of bytes to be read.

On devices capable of seeking, the `read` starts at a position in the file given by the file pointer associated with *fildes*. On return from `read`, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

On successful completion, `read` returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see `ioctl`(2) and `termio`(4)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When you try to read from an empty pipe (or FIFO special file), the following occurs:

- If `O_NDELAY` is set, the read returns a 0.
- If `O_NONBLOCK` is set, the read returns a –1.
- If `O_NDELAY` and `O_NONBLOCK` are both clear, the read blocks until data is written to the file or the file is no longer open for writing.

When you try to read a file associated with a tty that has no data currently available, the following occurs:

- If O_NDELAY is set, the read returns a 0.
- If O_NONBLOCK is set, the read returns a −1.
- If O_NDELAY and O_NONBLOCK are both clear, the read blocks until data becomes available.

When you try to read from a regular file that has mandatory file and record locking set (see chmod(2)), and a blocking write lock exists on the segment of the file to be read (that is, it is owned by another process):

- If either O_NDELAY or O_NONBLOCK is set, the read returns –1 and sets errno to EAGAIN.
- If O_NDELAY and O_NONBLOCK are both clear, the read sleeps until the blocking record lock is removed.

NOTES

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
PRIV_MAC_READ	The process is granted read permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If read completes successfully, a nonnegative integer is returned, indicating the number of bytes actually read; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The read system call fails if one of the following error conditions occurs:

Error Code	Description
EAGAIN	Mandatory file and record locking was set, O_NDELAY was set, and there was a blocking record lock.
EBADF	The *fildes* argument is not a valid file descriptor open for reading.
EBADF	The active security label is not greater than or equal to the security label of the file, and the process does not have appropriate privilege.
EDEADLK	The read was going to go to sleep and cause a deadlock situation to occur.
EFAULT	The *buf* argument points outside the allocated address space.
EINTR	A signal was caught during the read system call.

`EINVAL`	The call contains an argument that is not valid such as the dismounting of a nonmounted device, the mention of an undefined signal in `signal`(2) or `kill`(2), or the reading or writing of a file for which `lseek`(2) has generated a negative pointer. This error is also set by the math functions described in the (3) entries.
`ENOLCK`	The system record lock table was full; so the read could not go to sleep until the blocking record lock was removed.
`ENXIO`	During a read or write on a special file, a subdevice that does not exist or is beyond the limits of the device was referenced.

EXAMPLES

The following examples illustrate different uses of the `read` system call.

Example 1: This example shows a simple `read` request that reads 100 bytes sequentially from a regular data file on each execution of the `while` loop. A value of 0 returned by `read` indicates an EOF condition has been reached.

```
int fd, cnt;
char buf[100];

if ((fd = open("datafile", O_RDONLY)) == -1) {
     perror("Opening file datafile failed");
     exit(1);
}

while ((cnt = read(fd, buf, 100)) != 0) { /* read returning 0 means EOF */
     /* process data (cnt bytes) in buf here */
}

printf("EOF reached on file datafile.\n");
```

Example 2: This example shows how to use a `read` system call with an open pipe. Since the pipe is opened with the `O_NONBLOCK` flag set, `read` requests are not delayed when no data is available in the pipe to be read. (Typically, an empty pipe causes a `read` request to delay (block) until data arrives in the pipe.)

A value of 0 returned by `read` indicates an EOF condition has been reached; while a value of –1 returned means that no data is currently residing in the pipe to read.

```
int pfd, cnt, nbyte;
char pdata[256];

if ((pfd = open("named_pipe", O_RDONLY | O_NONBLOCK)) == -1) {
     perror("Opening named pipe named_pipe failed");
     exit(1);
}

if ((cnt = read(pfd, pdata, nbyte)) > 0) {
    /* process data (cnt bytes) from pipe in pdata here */
}
else {
     if (cnt == 0) {          /* read returning 0 means EOF */
          printf("EOF reached on pipe named_pipe.\n");
     }
     else {                   /* read returning -1 means no data now */
          /* no data currently available in pipe named_pipe -
             perform some other work and try again later */
     }
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `read` system call

SEE ALSO

`accept`(2), `chmod`(2), `creat`(2), `dup`(2), `fcntl`(2), `ioctl`(2), `kill`(2), `lseek`(2), `open`(2), `pipe`(2), `signal`(2), `socket`(2), `socketpair`(2)

`termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

reada – Performs asynchronous read from a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iosw.h>
#include <signal.h>

int reada (int fildes, char *buf, unsigned nbyte, struct iosw *status,
int signo);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The reada system call tries to read a specified number of bytes from a file into a specified buffer (see the read(2) man page). The system call returns directly, even when the data cannot be delivered until later.

The first three arguments of the reada system call are the same as the read(2) system call. The last two arguments enable you to notify the process when the request has completed.

The reada system call accepts the following arguments:

fildes Specifies a file descriptor. It is obtained from an accept(2), creat(2), dup(2), fcntl(2), open(2), pipe(2), socket(2), or socketpair(2) system call.

buf Points to the buffer into which the data is to be read.

nbyte Specifies the number of bytes to be read.

status Points to a iosw structure. This structure is defined in the usr/include/sys/iosw.h file. The *status* word has the following structure:

```
struct iosw   {
    uint   sw_flag     :1,
           sw_error    :31,
           sw_count    :32;
};
```

signo Specifies the signal that should be sent to indicate that the I/O transfer is complete. For a list of signals, see the signal(2) man page.

When a request completes, the status word is filled in, and if signo was nonzero, that signal is sent to the process. The sw_flag is always set on completion, sw_error may contain a system call error number (see the intro(2) man page), and sw_count contains the number of bytes actually moved. If a process issues reada on a slow device, such as a tty, and must be moved in memory to satisfy a brk(2) request, the reada will fail with EINTR.

When an attempt to read from a regular file with mandatory file and record locking set is made (see the `chmod`(2) man page), and a blocking write lock exists on the segment of the file to be read (that is, it is owned by another process), the following occurs:

- If either `O_NDELAY` or `O_NONBLOCK` is set, the read returns a –1, and sets `errno` to `EAGAIN`.
- If `O_NDELAY` and `O_NONBLOCK` are both clear, the read sleeps until the blocking record lock is removed.

There is a limit to the number of outstanding asynchronous I/O requests that a process may have active. If a process exceeds this limit, it is not rescheduled until one or more of the requests have completed.

The file position for reading or writing is always the file position at the time of the `reada` or `writea`(2) system call. The file's position is incremented at that time by *nbyte* bytes. In this way, `reada`, `writea`(2), and `lseek`(2) system calls can be interspersed, and the file position is incremented naturally.

To use asynchronous I/O effectively, several rules must be followed:

- All outstanding I/O requests must have their own status words.
- One or more signal numbers may be used for I/O completions, but each signal must have its own handling routine. Several outstanding requests may share a signal handling routine.
- When an I/O completion handler is entered, the status words under its control should be scanned for completed I/Os.
- As the status words are processed, they must be set to 0.
- At the end of I/O completion handling, the status words are rescanned for newly completed I/Os. If any are found, the signal handler loops back and processes the new completions; otherwise, the handler returns.

NOTES

The process must be granted read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If `reada` completes successfully, a nonnegative integer is returned, indicating the number of bytes remaining to be read; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `reada` system call fails if one of the following error conditions occurs:

Error Code	Description
`EAGAIN`	Mandatory file and record locking was set, `O_NDELAY` was set, and there was a blocking record lock.
`EBADF`	The *fildes* argument is not a valid file descriptor open for reading.
`EBADF`	The active security label of the process is not greater than or equal to the security label of the file, and the process does not have appropriate privilege.
`EDEADLK`	The read was going to go to sleep and cause a deadlock situation to occur.
`EFAULT`	The *buf* or *status* argument is not fully contained within the process address space.
`EINTR`	The process caught a signal during the `reada` system call.
`EINVAL`	The *signo* argument is not a valid signal number and not 0.
`ENOLCK`	The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.

FORTRAN EXTENSIONS

The `reada` system call can be called from Fortran as a function:

```
INTEGER fildes, buf(n), nbyte, istat, signo, READA, I
I = READA (fildes, buf, nbyte, istat, signo)
```

EXAMPLES

The following examples illustrate different ways of using the `reada` system call so that a read operation completes in parallel with other work in a user's process. Simpler solutions appear in the last two examples, which make use of additional calls.

Example 1: In this program, the `reada` request specifies delivery of a `SIGUSR1` signal on completion of the request.

The program uses the `pause`(2) system call to wait for the completion of the asynchronous read operation (that is, reception of the `SIGUSR1` signal). The `sigoff` library routine provides assurance that the `SIGUSR1` signal is not received before reaching the `pause`(2) request.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

struct iosw rdstat;

main()
{
     char buf[1000];
     int fd;
     void rdhdlr(int signo);

     signal(SIGUSR1, rdhdlr);

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     sigoff();      /* delay signal reception until pause() is reached */
     reada(fd, buf, 1000, &rdstat, SIGUSR1);  /* SIGUSR1 sent when
                                                 read completes */

     /* perform other work here in parallel with I/O completion */

     pause();       /* wait for read to complete - pause() calls sigon() */

     /* input data from reada now available in buffer buf */
}

void rdhdlr(int signo)
{
     signal(signo, rdhdlr);
     printf("reada read %d bytes\n", rdstat.sw_count);
     rdstat.sw_flag = 0;
}
```

Example 2: (Some system calls in the example are not supported on Cray MPP systems.) Unlike the program in example 1, this program uses the `recalla`(2) system call to wait for completion of the asynchronous input operation. The user's program is informed of the completion by reception of the `SIGUSR1` signal. While `recalla`(2) can wait for completion of multiple asynchronous I/O requests from multiple files, it only waits for one read operation in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>
#include <sys/param.h>

struct iosw rdstat;

main()
{
     char buf[1000];
     int fd;
     long mask[RECALL_SIZEOF];
     void rdhdlr(int signo);

     signal(SIGUSR1, rdhdlr);

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     RECALL_SET(mask, fd);                    /* set bit for fd in mask */

     reada(fd, buf, 1000, &rdstat, SIGUSR1);  /* SIGUSR1 sent when
                                                 read completes */

     /* perform other work here in parallel with I/O completion */

     recalla(mask);                           /* wait for read to complete */

     /* input data from reada now available in buffer buf */
}

void rdhdlr(int signo)
{
     signal(signo, rdhdlr);
     printf("reada read %d bytes\n", rdstat.sw_count);
     rdstat.sw_flag = 0;
}
```

Example 3: Unlike the programs in examples 1 and 2, this program does not have an I/O completion signal specified on the `reada` request. The program uses the `recall`(2) system call to wait for completion of the asynchronous read operation. While `recall`(2) can wait for completion of multiple asynchronous I/O requests from multiple files or even the same file, it only waits for one read operation in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

main()
{
     char buf[1000];
     int fd;
     struct iosw rdstat[1], *statlist[1];

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     reada(fd, buf, 1000, &rdstat[0], 0); /* no signal sent when
                                             read completes */
     statlist[0] = &rdstat[0];

     /* perform other work here in parallel with I/O completion */

     recall(fd, 1, statlist);              /* wait for read to complete */

     printf("reada read %d bytes\n", rdstat[0].sw_count);
     rdstat[0].sw_flag = 0;

     /* input data from reada now available in buffer buf */
}
```

SEE ALSO

`brk`(2), `chmod`(2), `intro`(2), `lseek`(2), `pause`(2), `read`(2), `recalla`(2), `recall`(2), `writea`(2)

`sigoff`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`readlink` – Reads value of a symbolic link

SYNOPSIS

`#include <unistd.h>`

`int readlink (char *`*path*`, char *`*buf*`, int` *bufsiz*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `readlink` system call places the contents of the symbolic link in a buffer of specified size. The contents of the link are not null terminated when returned.

The `readlink` system call accepts the following arguments:

path Specifies the contents of the symbolic link. That is, the path name of the file being referred.

buf Points to the buffer that contains the symbolic link.

bufsiz Specifies the buffer size in characters.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

The process must have read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to the component via the permission bits and access control list.
`PRIV_MAC_READ`	The calling process is granted read permission to the file via the security label.
`PRIV_MAC_READ`	The process is granted search permission to the component via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix. If the `PRIV_SU` configuration option is enabled, the super user is granted read permission to the file via the security label.

RETURN VALUES

If `readlink` completes successfully, the count of characters placed in the buffer is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `readlink` system call fails and the buffer is unchanged if one of the following error conditions occurs:

Error Code	Description
`EACCES`	Search permission is denied for a component of the path prefix of *path*.
`EACCES`	The process is not granted read permission to the file via the security label and does not have appropriate privilege.
`EFAULT`	The *path* or *buf* argument extends outside the process allocated address space.
`EINVAL`	The specified file is not a symbolic link.
`EINVAL`	The *path* argument contained a byte with the high-order bit set.
`EIO`	An I/O error occurred during a read from or write to the file system.
`EMLINK`	Too many symbolic links were encountered in translating *path*.
`ENAMETOOLONG`	The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.
`ENOENT`	The specified file does not exist.

EXAMPLES

This example shows how to use the `readlink` system call to retrieve the path name of a file for which a symbolic link is targeting.

First, a `symlink`(2) system call is used to create a symbolic link. (Two arguments must be supplied to this program; the first is the path name of an existing file, which is the target of the symbolic link, and the second is the new link.) The `readlink` request then returns the path name of the target of the new link (that is, the value of the first argument, `argv[1]`).

```
#include <unistd.h>

main(int argc, char *argv[])
{
     char buf[1024];

     if (symlink(argv[1], argv[2]) == -1) { /* argv[1] -> existing file */
          perror("symlink failed");         /* argv[2] -> new link      */
     }

     readlink(argv[2], buf, 1024);          /* buf should contain argv[1]
                                               string */

     printf("Symbolic link contains => %s\n", buf);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `readlink` system call

SEE ALSO

`lstat`(2), `stat`(2), `symlink`(2)

NAME

`recall`, `recalls` – Waits for I/O completions

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iosw.h>

int recall (int fildes, int cnt, struct iosw **list);

int recalls (int cnt, struct iosw **list);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `recall` system call provides a means to wait for any of a specified set of asynchronous I/O requests to complete. Each element in *list*, if it is not null, points to an I/O completion status word. When the completion bit is set in any of the specified status words, the system call returns. Any null entries in *list* are ignored. The arguments are as follows:

fildes Specifies a file descriptor. It is obtained from a `creat`(2), `dup`(2), `fcntl`(2), `open`(2), or `pipe`(2) system call or socket descriptor obtained from a call to the `socket`(2) system call.

cnt Specifies the number of elements in *list*.

list Points to an array of pointers to asynchronous I/O status structures of type `struct iosw`.

When calling `recall`, all status structures in *list* must correspond to I/O requests made for file descriptor *fildes*. The `recalls` system call has the same effect as `recall`, but the restriction that all status structures in *list* correspond to I/O requests for just one file descriptor is relaxed. Users are encouraged to use `recall` because the *fildes* argument permits the `procstat`(1) command to gather more complete statistics.

RETURN VALUES

If `recall` or `recalls` completes successfully, the value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `recall` or `recalls` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The request *list* is not fully contained within the process address space.
`EINTR`	A signal was caught during a wait for an I/O completion.

`EINVAL` The *cnt* argument is not a valid size. Implementation-defined limits exist on the maximum size of this list.

EXAMPLES

The following example shows how to use the `recall` system call to wait for completion of an asynchronous read operation so that the operation is performed in parallel with other work in a user's process.

In this program, the `reada`(2) request does not include an I/O completion signal. The `recall` request waits for the read operation to complete. Although `recall` can wait for completion of multiple asynchronous I/O requests from multiple files or even the same file, it waits for only one read operation in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

main()
{
     char buf[1000];
     int fildes;
     struct iosw rdstat[1], *statlist[1];

     if ((fildes = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     reada(fildes, buf, 1000, &rdstat[0], 0); /* no signal sent when
                                                 read completes */
     statlist[0] = &rdstat[0];

     /* perform other work here in parallel with I/O completion */

     recall(fildes, 1, statlist);         /* wait for read to complete */

     printf("reada read %d bytes\n", rdstat[0].sw_count);
     rdstat[0].sw_flag = 0;

     /* input data from reada now available in buffer buf */
}
```

SEE ALSO

`creat`(2), `dup`(2), `fcntl`(2), `listio`(2), `open`(2), `pipe`(2), `reada`(2), `recalla`(2), `socket`(2), `writea`(2)

`procstat`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`recalla` – Waits for I/O completion(s)

SYNOPSIS

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/iosw.h>

int recalla (long mask[RECALL_SIZEOF]);

RECALL_INIT (mask);
RECALL_CLR (mask, fd);
RECALL_SET (mask, fd);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `recalla` system call waits for the completion of one or more I/O requests on files specified by *mask* that were previously initiated by a `reada`(2) or `writea`(2) system call.

It accepts the following argument:

mask Specifies a left-justified bit array in which each bit corresponds to a file descriptor. The bit array is an array of size `RECALL_SIZEOF` where each array element is of type `long`.

If any of the files to which *mask* points are not busy, control is returned immediately. If all of the files to which *mask* points are busy, the process is blocked until at least one of the file's I/O requests completes.

The caller must check the status word associated with the files in the mask to ensure completion.

The following macros are defined in the `sys/iosw.h` file. In these macros, *mask* specifies the *mask* argument used in the `recalla` call, and *fd* specifies a file descriptor.

`RECALL_INIT` (*mask*)	Clears all bits in *mask*.
`RECALL_CLR` (*mask*, *fd*)	Clears bit corresponding to *fd* in *mask*.
`RECALL_SET` (*mask*, *fd*)	Sets bit corresponding to *fd* in *mask*.

RETURN VALUES

If `recalla` completes successfully, the value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `recalla` system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The request *mask* is not fully contained within the process address space.
EINTR	A signal was caught during a wait for an I/O completion.

EXAMPLES

The following example shows how to use the `recalla` system call to wait for completion of an asynchronous read operation so that the operation is performed in parallel with other work in a user's process.

In this program, the `reada`(2) request specifies delivery of a `SIGUSR1` signal on completion of the request. The `recalla` system call waits for the completion of the read operation. The user's program is informed of the completion by the reception of the `SIGUSR1` signal. While `recalla` can wait for completion of multiple asynchronous I/O requests from multiple files, it only waits for one read request in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>
#include <sys/param.h>

struct iosw rdstat;

main()
{
     char buf[1000];
     int fd;
     long mask[RECALL_SIZEOF];
     void rdhdlr(int signo);

     signal(SIGUSR1, rdhdlr);

     if ((fd = open("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }

     RECALL_SET(mask, fd);                    /* set bit for fd in mask */

     reada(fd, buf, 1000, &rdstat, SIGUSR1);  /* SIGUSR1 sent when
                                                 read completes */

     /* perform other work here in parallel with I/O completion */

     recalla(mask);                          /* wait for read to complete */
```

```
        /* input data from reada now available in buffer buf */
    }

    void rdhdlr(int signo)
    {
        signal(signo, rdhdlr);
        printf("reada read %d bytes\n", rdstat.sw_count);
        rdstat.sw_flag = 0;
    }
```

SEE ALSO

listio(2), open(2), reada(2), recall(2), writea(2)

NAME

`recv`, `recvfrom`, `recvmsg` – Receives a message from a socket

SYNOPSIS

All Cray Research systems:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/socket.h>

int recv (int s, char *buf, int len, int flags);

int recvfrom (int s, char *buf, int len, int flags,
struct sockaddr *from, int *fromlen);
```

Cray PVP systems:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/socket.h>

int recvmsg (int s, struct msghdr *buf, int flags);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `recv`, `recvfrom`, and `recvmsg` system calls receive a message (*buf*) from a socket.

You can use a `recv` call only on a connected socket. You can use `recvfrom` or `recvmsg` on either a connected or unconnected socket. The `recvmsg` system call uses the same `msghdr` structure as the `sendmsg`(2) system call to minimize the number of directly supplied arguments. For more information, see the `connect`(2) and `send`(2) man pages.

The `recv`, `recvfrom`, and `recvmsg` system calls accept the following arguments:

s Specifies the descriptor of the socket from which messages are received. Descriptor is returned when a socket is created by the `socket`(2) or `socketpair`(2) system call; `socketpair`(2) uniquely identifies the socket's access path.

buf Points to the address of a buffer into which received messages are placed.

len Specifies the length of the buffer pointed to by the *buf* argument.

flags Allows the caller to control the reception of messages. The argument value is formed by performing an OR operation on one or more of the following values:

`MSG_OOB` Process out-of-band data.

`MSG_PEEK` Peek at incoming message without removing message from the socket.

`MSG_DONTROUTE`	Send without using routing tables.
`MSG_EOR`	Data sent completes record.
`MSG_TRUNC`	Data discarded before delivery.
`MSG_CTRUNC`	Control data lost before delivery.
`MSG_WAITALL`	Wait for full request or error.

The `recvfrom` and `recvmsg` system calls can be used to receive data on a socket whether or not it is in a connected state. The `recvfrom` system call accepts the following additional arguments allowing the caller to specify where the sender's address should be recorded:

from Specifies the address of a sockaddr structure that the operating system will use to record the address of the message sender.

fromlen Specifies the address of an integer that the operating system uses to record the length of the sender's address, which is recorded in *from*.

If no messages are available at the socket, the receive call waits for a message to arrive; however, if the socket is nonblocking (set by using an `ioctl`(2) system call with `FIONBIO` (see `socket`(2)), a value of –1 is returned, and the external variable `errno` is set to `EWOULDBLOCK`. Use the `select`(2) call to determine when data arrives.

The `recvmsg` system call uses the same `msghdr` structure which is defined in the `sys/socket.h` file. This structure has the following form:

```
struct msghdr {
        caddr_t    msg_name;            /* optional address */
        u_int      msg_namelen;         /* size of address */
        struct     iovec *msg_iov;      /* scatter/gather array */
        u_int      msg_iovlen;          /* # elements in msg_iov */
        caddr_t    msg_control;         /* ancillary data, see below */
        u_int      msg_controllen;      /* ancillary data buffer len */
        int        msg_flags;           /* flags on received message */
};
```

In this structure, `msg_name` and `msg_namelen` describe the source address for `recvmsg` or the destination address for `sendmsg`(2). If no names are desired or required, `msg_name` is given as a null pointer. The `msg_name` and `msg_namelen` arguments operate similarly to the `recvfrom` *from* and *fromlen* arguments, and the `sendto`(2) *to* and *tolen* arguments.

The `msg_iov` and `msg_iovlen` arguments describe an array of buffer descriptors. The `msg_iov` argument points to an array of structure `iovec`, which is defined as follows:

```
struct iovec   {
    caddr_t    iov_base;
    int        iov_len;
};
```

Each `iovec` entry specifies the base address and length of an area in memory from which data must be read or to which data must be written.

The *iov_len* argument specifies the number of structure `iovec` entries in the array to which `msg_iov` points. `recvmsg` and `sendmsg`(2) always process the entire `iov_len` bytes of one `iovec` structure before proceeding to the next.

The `msg_control` argument, which has length `msg_controllen`, is a buffer for other protocol control-related messages or other miscellaneous ancillary data. The messages are of the following form:

```
struct cmsghdr {
        u_int   cmsg_len;       /* data byte count, including hdr */
        int     cmsg_level;     /* originating protocol */
        int     cmsg_type;      /* protocol-specific type */
/* followed by
        u_char  cmsg_data[]; */
};
```

For `recvmsg`, the `msg_controllen` argument is a value-result parameter, which is initialized to the size of the `msg_control` argument, and it displays the number of bytes of control information returned.

Open file descriptors are now passed as ancillary data for `AF_UNIX` domain sockets (for example, `cmsg_level` is `SOL_SOCKET` and `cmsg_type` is `SCM_RIGHTS`). This feature is disabled on systems that are configured to support nonzero security labels.

The `msg_flags` argument is set on return using a method that includes some of the same values that are specified for the *flags* parameter to a `recv` system call. The returned value `MSG_EOR` indicates end-of-record, `MSG_TRUNC` indicates that some trailing datagram data was discarded, and `MSG_CTRUNC` indicates that some control data was discarded because of lack of space. `MSG_OOB` is returned to indicate that expedited data was received.

NOTES

If the `SOCKET_MAC` option is enabled, the active security label of the process must be greater than or equal to the security label of the socket. `SOCKET_MAC` is part of TCP/IP configurable feature variables list in `uts/cf/Nmakefile`. For more information, see the `connect`(2) man page.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_READ`	When the `SOCKET_MAC` is enabled, the process may override the security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user may override security label restrictions when the `SOCKET_MAC` option is enabled.

If *s* is an Internet-domain socket and the `recvfrom` or `recvmsg` system call is used, the `sin_addr` and `sin_port` fields of the sending socket name identify only the socket at the other end of the connection, not the remote process or remote user. Additional knowledge is required to interpret those fields. For example, if the `sin_addr` field designates another UNICOS system, a `sin_port` value of less than 1024 indicates a connection with trusted software (for example, `rlogin`(1B)), which may include additional identity information in its protocol data stream. If it is necessary to identify the actual user associated with the socket, the communicating peers must agree in advance on a method, such as the sender placing its `sin_port` value in a protected file accessed through NFS (or other means) by the receiver.

Because no sender name information can be obtained from a UNIX-domain socket, the other end of the connection cannot be identified except to the extent that additional authentication techniques are used. Although no identity-based access controls restrict use of `connect`(2) or `sendto`(2) for a UNIX-domain socket, such a socket can be created in a directory to which execute (search) access is restricted. This limits the ability of other processes to connect to the socket. Alternatively, the listening process could place a random value or secret password in a protected file and require the value or password be included in all messages it accepts; this ensures that only users with access to that file can send valid messages. For both Internet-domain and UNIX-domain, this authentication requires explicit action on the part of the receiver.

RETURN VALUES

If `recv`, `recvmsg`, or `recvfrom` completes successfully, the number of characters received is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `recv`, `recvfrom`, or `recvmsg` system call fails if one of the following conditions occurs:

Error Code	**Description**
`EACCES`	Permission is denied (because of a security violation).
`EACCES`	If the `SOCKET_MAC` option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
`EBADF`	Descriptor *s* is not valid.
`EFAULT`	Data is specified to be received into a nonexistent or protected part of the process address space.
`EINTR`	Receive operation is interrupted by delivery of a signal before any data is available for the receive.
`EMSGSIZE`	The `msg_iovlen` field is greater than or equal to the `MSG_MAXIOVLEN` parameter (defined in the `sys/socket.h` file).
`EINVAL`	No out-of-band data is available when the `MSG_OOB` flag is specified.
`ENOTCONN`	Socket is not connected.
`ENOTSOCK`	Descriptor *s* is not a socket.

`EWOULDBLOCK` Socket is marked nonblocking, and the receive operation would block.

FILES

`/etc/config/spnet.conf`	Network access list file
`/usr/include/sys/socket.h`	Contains definitions related to sockets, types, address families, and options
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/sys/uio.h`	Contains user I/O structures and definitions

SEE ALSO

`accept`(2), `connect`(2), `ioctl`(2), `select`(2), `send`(2), `socket`(2), `socketpair`(2)

`rlogin`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

NAME

`rename` – Changes the name of a file

SYNOPSIS

```
#include <stdio.h>
int rename (const char *old, const char *new);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `rename` system call changes the name of a file.

The `rename` system call accepts the following arguments:

old Points to the path name of the file to be renamed.

new Points to the new path name of the file.

If the *old* argument and the *new* argument both refer to links to the same existing file, the `rename` system call returns successfully and performs no other action.

If the *old* argument points to the path name of a file that is not a directory, the *new* argument does not point to the path name of a directory. If the link specified by the *new* argument exists, it is removed and *old* is renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory that contains *old* and the directory that contains *new*.

If the *old* argument points to the path name of a directory, the *new* argument points to the path name of a file that is a directory. If the directory specified by the *new* argument exists, it shall be removed and *old* renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. If *new* specifies an existing directory, it must be an empty directory.

The *new* path name does not contain a path prefix that names *old*. Write access permission is required for the directory that contains *old* and the directory that contains *new*. If the *old* argument points to the path name of a directory, write access permission is required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link specified by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file is freed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before `rename` returns, but the removal of the file contents is postponed until all references to the file are closed.

Upon successful completion, `rename` marks for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

NOTES

Under UNICOS, `rename` is implemented as a system call, but the `rename`(3C) function is also defined to be a part of the ANSI Standard C library. For this reason, this documentation appears both here and in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080.

The process must be granted search permission to every component of each path prefix via the permission bits and access control list. The process must be granted search permission to every component of each path prefix via the security label.

The process must be granted write permission to each path's parent directory via the permission bits and access control list. The process must be granted write permission to each path's parent directory via the security label.

If *old* is a directory, and *new*'s parent directory is not the same as *old*'s parent directory, the process must be granted write permission to *old* via the permission bits and access control list. If *old* is a directory and *new*'s parent directory is not the same as *old*'s parent directory, the process must be granted write permission to *old* via the security label.

If *new* already exists, the process must be granted write permission to *new* via the permission bits and access control list. If *new* already exists, the process must be granted write permission to *new* via the security label.

If `FSETID_RESTRICT` is enabled, the set-user-ID and set-group-ID bits are cleared if the file is renamed across file systems, unless the process has appropriate privilege.

A process with the effective privileges shown are granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of each path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to each path's parent directory *old*, and *new* (if it already exists) via the permission bits and access control list.
`PRIV_FSETID`	If `FSETID_RESTRICT` is enabled, set-user-ID and set-group-ID bits are not cleared access file systems.
`PRIV_MAC_READ`	The process is granted search permission to every component of each path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to each path's parent directory *old*, and *new* (if it already exists) via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of each path prefix and is granted write permission to each path's parent directory. The super user is granted write permission to *old* and *new* (if it exists). The super user or a process with the `suidgid` permission can override the restriction enabled by the `FSETID_RESTRICT` system configuration option.

RETURN VALUES

If `rename` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `rename` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	A component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or, write permission is required and is denied for a directory pointed to by the *old* or *new* argument.
`EBUSY`	The directory named by *old* or *new* cannot be renamed because it is being used by the system or another process and the implementation considers this to be an error.
`EEXIST` or `ENOTEMPTY`	The link named by *new* is a directory containing entries other than dot and dot-dot.
`EINVAL`	The *new* directory path name contains a path prefix that names the *old* directory.
`EISDIR`	The *new* argument points to a directory and the *old* argument points to a file that is not a directory.
`ENAMETOOLONG`	The length of the *old* or *new* argument exceeds `{PATH_MAX}`, or a path name component is longer than `{NAME_MAX}` when `{_POSIX_NO_TRUNC}` is in effect.
`ENOENT`	The link named by the *old* argument does not exist or either *old* or *new* points to an empty string.
`ENOSPC`	The directory that should contain *new* cannot be extended.
`ENOTDIR`	A component of either path prefix is not a directory; or the *old* argument names a directory and the *new* argument names a nondirectory file.
`EROFS`	The requested operation requires writing in a directory on a read-only file system.
`EXDEV`	The links named by *new* and *old* are on different file systems and the implementation does not support links between file systems.

FILES

`/usr/include/sys/stdlib.h` Contains C prototype for the `rename` system call

SEE ALSO

`link`(2), `rmdir`(2), `unlink`(2)

`rename`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`resch` – Reschedules a process

SYNOPSIS

```
#include <unistd.h>
void resch (void);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `resch` system call causes a process to be rescheduled by logically placing it at the end of the queue of processes that can run.

If the process is a thread-process with its wakeup flag set to 0, it is suspended until the wakeup flag is set to a nonzero value. This is used by the microtasking library to activate and deactivate processes.

RETURN VALUES

None

FILES

`/usr/include/unistd.h`	Contains C prototype for the `resch` system call

SEE ALSO

`fork`(2), `thread`(2)

NAME

`restart` – Restarts a process, multitask group, or job

SYNOPSIS

```
#include <sys/restart.h>
int restart (char *path, long flags);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `restart` system call validates, loads, and restarts the process, multitask group, or job defined in the specified restart file as created by the `chkpnt`(2) system call.

It accepts the following arguments:

path Specifies path name of the restart file containing the process, multitask group, or job (or interactive session) to be recovered.

flags Identifies optional `restart` actions. The flags present in this field are OR'ed together to define the optional actions to be performed by `restart`. The optional actions are as follows:

`RESTART_FORCE` Forces recovery even when one or more of the files referred to by processes saved in the restart file have been changed.

`RESTART_PAG` Allows the restarted processes to inherit the Distributed Computing Environment (DCE) credentials from the process that calls `restart` rather than using the credentials stored in the restart file.

`RESTART_PTRACE` Restarts and traces the process (see `ptrace`(2)). If the `RESTART_PTRACE` flag is set, the restart file must describe a process or multitask group, not a job. If the restart is successful, the restarted process is recovered as though it had executed a `ptrace`(2) system call. When restarting a multitask group, it is as though the eldest process of the multitask group executed the `ptrace`(2) system call.

`RESTART_SUSPEND`
Restarts all the recovered processes in a suspended state (see `suspend`(2) and `resume`(2)).

`RESTART_WAIT` Makes the restarted process, if it is interactive, the foreground process.

The eldest process of each multitask group receives a `SIGRECOVERY` signal on restoration (see `signal`(2)). By default, the `SIGRECOVERY` signal is ignored, but processes expecting to be checkpointed and restored successively can elect to catch this signal; thereby, they can perform any special actions needed for their proper recovery.

Processes with open pipes can be checkpointed and restarted if their pipe connections do not go outside the job or multitask group being checkpointed. For a process with open pipes to have been checkpointed, all of its pipe connections must have terminated with processes also included in the restart file.

Processes with open files that reside on NFS file systems can be checkpointed and restarted. To restart a process with open NFS files, the NFS file systems on which the files reside have to be mounted, unless the NFS file systems are managed by the automounter. In this case, the automounter will try to remount the file systems automatically.

Processes with open files that reside on Distributed File System (DFS) file systems can be checkpointed and restarted. The following conditions must exist in order for a user to restart a process with an open DFS file.

- The DFS client must be running on the local host.
- The DFS server must be running on the host where the file resides.

Access to DFS files is controlled by a user's DCE credentials as opposed to user identification (UID) and group identification (GID). DFS credentials consist of Kerberos tickets stored in a special file. When a process is checkpointed, a reference to these credentials is stored in the restart file. The credentials must be present and valid when `restart` is performed. If the credentials are no longer present or have expired, accesses to DFS files that are performed after the `restart` call will appear to be from the UID –2.

The `RESTART_PAG` flag allows the reference to the original credentials to be replaced by a reference to a new set of credentials. Using this flag allows processes that access DFS files to be restarted after their original credentials have been deleted or have expired.

Processes with unlinked files can be checkpointed and restarted if the total of the size of all unlinked files in use by the target process set is within the size limit established by the system administrator. See the system variable `MAX_UNLINKED_BYTES` in the `/usr/src/uts/c1/cf/config.h` file to see the site local definition.

On systems with SSD solid-state storage devices, processes that are using secondary data segments (SDS) can be checkpointed and restarted if sufficient disk space is available and can contain an image of the process' SDS area within the restart file. An `ENOSDS` error may occur at restart time if the SDS area available at that time is less than what was in use at checkpoint time. The `ENOSDS` error means that `restart` must be retried at a later time when sufficient SDS space is available.

Processes using online tape files cannot be checkpointed or restarted.

NOTES

The following restrictions apply to processes and jobs (including interactive sessions) that are to be restarted:

- All files that a process was using when it was checkpointed must be present when the process is restarted. These files include all open files, any shared-text executables that the process was using (such as shells), and the present working directory. In the restart file, each of these files is identified by its inode number and the minor number of the file system. If either changes, the `restart` system call fails, and the call returns an `EFILERM` error. For example, if a file system is restored by `/etc/restore`, any process that was using files on that file system and was checkpointed before the restore, will fail to restart. After the restore, each file on the file system has a different inode number than it did when the process was checkpointed.

- Only a process with appropriate privilege may checkpoint or restart another user’s job or process.
- Processes using online tapes cannot be checkpointed or restarted.
- Processes using shared memory segments (CRAY T90 series systems only) cannot be checkpointed or restarted.
- Whenever any process is recovered from a restart file, all its multitask sibling processes are also recovered. Thus, when `restart` is invoked to perform recovery from a process restart file (a restart file that does not define an entire job), it is still possible for several processes to be recovered, because all the multitask siblings of the original target process must also be restored.
- All processes recovered by `restart` retain their original attributes, such as process ID (PID), parent process ID (PPID), process group ID (PGRP), and job ID (JID). The only possible exceptions to this rule concern the process attributes of PGRP and PPID of the oldest restarted process.
- The exception to the *pgrp* conservation rule occurs only when one multitask group is recovered from a process restart file. If the *pgrp* of the recovered multitask group is found to be nonzero and not equal to the *pid* of a process in the group, the *pgrp* of the recovered group is set to the *pgrp* of the caller.
- If a user attempts to copy a restart file, the `restart` system call fails.
- If a user attempts to move a restart file to a different file system, the `restart` system call fails.
- If an interactive session is checkpointed and later recovered with a `restart` system call, each process that is part of the session that performed the `restart` system call is sent a `SIGHUP` signal to indicate that the connection hung up. The call assigns the pseudo tty of the calling session to the restarted session.

A process with the effective privilege shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_FOWNER`	The process is considered the owner of the specified file.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the restart file via the security label.
`PRIV_POWNER`	The calling process is considered the owner of the session being restarted.

If the `PRIV_SU` configuration option is enabled, the super user is considered the owner of the restart file and is granted access to the restart file. The super user is considered the owner of the session being restarted.

RETURN VALUES

If `restart` completes successfully, the PID (JID) of the recovered process (job) is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `restart` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A component of the restart file path prefix denies search permission.
EACCES	The restart file is not owned by the caller, and the caller does not have appropriate privilege.
EACCES	The caller's active security label did not dominate that of the restart file.
EAGAIN	The system-imposed limit on the total number of processes in the system (NPROC) would be exceeded by the recovery of all processes from the restart file.
EAGAIN	The system-imposed limit on the total number of processes in the system allocated to one user (CHILD_MAX) would be exceeded by the recovery of all processes from the restart file.
EAGAIN	The restart file contains recovery information for an entire job, and the maximum number of jobs allowed to exist in the system (NJOB) at any one time already exist.
EAGAIN	The multitask group or process defined in the restart file could not be recovered, because a job I/O quota would be exceeded.
EBUSY	One or more of the processes to be recovered from the restart file has a process ID that is already allocated to an existing process in the system.
EBUSY	The restart file contains recovery information for an entire job, and the job ID of the job to be recovered is already allocated to an existing job in the system.
EDEADLK	The reapplication of record lock(s) owned by the process(es) to be restarted would result in a deadlock situation.
EFAULT	The *path* argument points outside the allocated process address space.
EFILECH	One or more files referenced in the restart file have been changed, and the RESTART_FORCE flag was not set.
EFILECH	One or more files referenced in the restart file have changed either user or group ownership. This situation cannot be overridden by the RESTART_FORCE flag.
EFILERM	One or more files referenced in the restart file are no longer present.
EFILERM	One or more files referenced in the restart file reside on an NFS file system that is not mounted.
EFILERM	A DFS file cannot be located. Either the file has been removed, or the DFS server holding it is either down or unreachable.
EINVAL	The `restart` system call was invoked with the RESTART_PTRACE flag, and the restart file described a job rather than a process or multitask group.
ENFILE	The system inode or file table is full.

`ENODEV`	The DFS client is currently not running and needs to be started before `restart` can proceed.
`ENOENT`	The specified restart file does not exist.
`ENOEXEC`	The restart file path name does not refer to a valid restart file.
`ENOLCK`	One or more of the processes to be restored owned record locks at checkpoint time (see `fcntl`(2) and `lockf`(3C)), and not enough record locks are available to complete recovery.
`ENOMEM`	There is not enough main memory or swap space to complete the recovery.
`ENOSDS`	Insufficient SDS space is available to complete the recovery.
`ENOSPC`	Insufficient free file space exists to re-create unnamed pipes previously in use by one or more of the processes to be recovered.
`ENOSPC`	Insufficient free file space exists to re-create unlinked regular files previously in use by one or more of the processes to be recovered.
`ENOTDIR`	A component of the restart file path prefix is not a directory.
`ENOTTY`	One or more of the processes to be recovered had a controlling tty, and the caller has no controlling tty.
`EQACT`	A file or inode quota limit was reached for the current account ID.
`EQGRP`	A file or inode quota limit was reached for the current group ID.
`EQUSR`	A file or inode quota limit was reached for the current user ID.
`ERFLOCK`	Record lock(s) owned by the process(es) to be restarted could not be reapplied because record lock(s) owned by currently existing process(es) have one or more of the target file regions already locked.

EXAMPLES

The following example shows how to use the `restart` system call to recover a checkpointed process or job. The path name of the checkpoint/restart file, `argv[1]`, is supplied as the only argument to this program.

(This system call is not used frequently by users because the `restart`(1) command provides similar functionality.)

```
main(int argc, char *argv[])
{
     int id;

     if ((id = restart(argv[1], 0)) == -1) {
          perror("restart failed");
          exit(1);
     }

     printf("pid (jid) of recovered process (job) = %d\n\n", id);
     system("ps");  /* view recovered processes in ps(1) display */
}
```

FILES

`/usr/include/sys/restart.h`	Contains the optional `restart` actions
`/usr/src/uts/c1/cf/config.h`	Contains the system variable `MAX_UNLINKED_BYTES`

SEE ALSO

`chkpnt`(2), `chmod`(2), `chown`(2), `creat`(2), `fcntl`(2), `open`(2), `ptrace`(2), `resume`(2), `signal`(2), `suspend`(2), `_tfork`(2)

`chkpnt_util`(1), `restart`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`lockf`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`rmdir` – Removes a directory

SYNOPSIS

```
#include <unistd.h>
int rmdir (const char *path);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `rmdir` system call removes the directory specified by a path name. The directory must not have any entries other than "." and "..".

The `rmdir` system call accepts the following argument:

path Points to the path name of the directory.

To remove a directory that has the "sticky" bit set, the process must be the owner of that directory.

NOTES

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the parent directory, the active security label of the process must equal the security label of the parent directory.

The process must be granted write permission to the directory being removed via the security label. That is, the active security label of the process must equal the security label of the directory being removed.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the parent directory via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_FOWNER`	The process is considered the owner of a directory that has the "sticky" bit set.
`PRIV_MAC_READ`	The calling process is granted search permission to every component of the path prefix via the security label.

`PRIV_MAC_WRITE`	The process is granted write permission to the parent directory via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the directory being removed via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix. If the `PRIV_SU` configuration option is enabled, the super user is granted write permission to the parent directory and to the directory being removed. The super user is considered the owner of a directory that has the "sticky" bit set.

RETURN VALUES

If `rmdir` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The specified directory is removed unless one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Search permission is denied for a component of the path prefix.
`EACCES`	Write permission is denied on the directory containing the directory to be removed.
`EACCES`	The process is not granted write permission to the directory being removed via the security label, and the process does not have appropriate privilege.
`EACCES`	Directory label does not dominate the active security label of the process.
`EACCES`	Parent directory label does not equal the active security label of the process.
`EBUSY`	The directory to be removed is the mount point for a mounted file system.
`EEXIST`	The directory contains entries other than those for "." and "..".
`EFAULT`	The *path* argument points outside the process' allocated address space.
`EINVAL`	The current directory may not be removed.
`EINVAL`	The "." entry of a directory may not be removed.
`EIO`	An I/O error occurred during the access of the file system.
`EMLINK`	The directory has been linked. Use `unlink`(2) to remove the directory.
`ENOENT`	The specified directory does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The directory has the "sticky" bit set and the process is not the owner.
`EROFS`	The directory entry to be removed is part of a read-only file system.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `rmdir` system call

SEE ALSO

`mkdir`(2), `unlink`(2)

`mkdir`(1), `rm`(1), `rmdir`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`rmfacl` – Removes an access control list from a file

SYNOPSIS

```
#include <sys/acl.h>
int rmfacl (char *fname);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `rmfacl` system call removes the access control list (ACL) from a file. A `rmfacl` request is allowed only for a process with an active `secadm` category, a process executing on behalf of the file owner, or a process with appropriate privilege. If the process is not a member of the owning group of the file, the set-group-ID mode bit of the file is cleared unless the process has appropriate privilege. If the `FSETID_RESTRICT` system configuration parameter is enabled, the set-user-ID and set-group-ID mode bits of a file are cleared unless the process has appropriate privilege.

The `rmfacl` system call accepts the following argument:

fname Specifies the file from which the ACL is removed.

NOTES

Errors are recorded in the security log if discretionary access logging is enabled.

The functionality provided by this system call is also provided by the `setfacl`(2) system call.

The process must have write permission to the file via the security label. That is, the active security label of the process must equal the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to the component via the permission bits and ACL.
`PRIV_FOWNER`	The process is considered the file’s owner.
`PRIV_FSETID`	The process is allowed to set an ACL on a file whose mode includes the set-user-ID or set-group-ID mode bit.
`PRIV_MAC_READ`	The process is granted search permission to the component via the security label.

`PRIV_MAC_WRITE`	The process is granted write permission to a component of the path prefix via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the file via the security label. The super user is considered the file owner and is allowed to set an ACL on a file whose mode includes the set-user-ID or set-group-ID mode bit.

RETURN VALUES

If `rmfacl` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `rmfacl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`EFAULT`	The *fname* argument points outside the process address space.
`EMANDV`	The process does not have write permission to the file via the security label and does not have appropriate privilege.
`ENAMETOOLONG`	The supplied file name is too long.
`ENOACL`	The specified file does not have an ACL or its ACL is corrupted.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EOWNV`	The process is not the file owner and does not have appropriate privilege.

FILES

`/usr/include/sys/acl.h`	Contains C prototype for the `rmfacl` system call

SEE ALSO

`getfacl`(2), `setfacl`(2)

`spacl`(1), `spclr`(1), `spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`acl`(5), `slog`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`schedv` – Sets memory scheduling parameters

SYNOPSIS

```
#include <sys/schedv.h>
int schedv (int svar, struct schedvar *svartab);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `schedv` system call gets and sets the system memory scheduling (`schedvar`) structure. It accepts the following arguments:

svar Specifies the command type. *svar* can be one of the following:

- `SVAR_GET` Transfers system `schedvar` table to the specified *svartab* address.
- `SVAR_SET` Transfers the `schedvar` table specified by *svartab* to the system `schedvar` table. The caller must fill in the `sv_magic` and `sv_size` fields in the `schedvar` structure with the `SV_MAGIC` constant and the size of the `schedvar` structure. Only a process with appropriate privilege can specify this command type.

svartab Points to the `schedvar` structure.

The `schedvar` structure includes the following members:

```
int     sv_memhog;         /* Size of a "big" process in clicks     */
time_t  sv_cpuhog;         /* Utime ticks used by CPU-bound proc.   */
int     sv_hog_max_mem;    /* Max clicks allotted to "hog" procs    */
float   sv_fit_boost;      /* Best fit boost given to in-core proc  */
                           /* if bigger than proc coming in         */
int     sv_thrash_inter;   /* Thrash interval in seconds            */
int     sv_thrash_blks;    /* Thrash blocks per interval            */
float   sv_mfactor_in;     /* Memory size factor - loaded procs     */
float   sv_mfactor_out;    /* Memory size factor - swapped procs    */
float   sv_tfactor_in;     /* Time factor - loaded procs            */
float   sv_tfactor_out;    /* Time factor - swapped procs           */
                           /* tfactor's are multiplied against time */
                           /* of residence.                         */
float   sv_pfactor_in;     /* Priority factor - loaded procs        */
float   sv_pfactor_out;    /* Priority factor - swapped procs       */
float   sv_nfactor_in;     /* Nice factor - loaded procs            */
float   sv_nfactor_out;    /* Nice factor - swapped procs           */
int     sv_max_outage;     /* Maximum time in seconds for which a   */
                           /* swapped process will be passed over   */
```

```
                                /* during memory-tight situations.       */
int      sv_flags;              /* Behavior modification flags           */
                                /* Used for things such as interactive   */
                                /* preferred, etc. See defines in        */
                           /* schedv.h for list of flags            */
time_t   sv_packtime;           /* The time, in clocks, between attempts */
                                /* slide processes down to pack memory.  */
float    sv_kfactor_in;         /* 0'th polynomial term - incore procs   */
float    sv_kfactor_out;        /* 0'th polynomial term - swapped procs  */
float    sv_gfactor0_in;        /* Guaranteed residence factor 0         */
                                /*                   - loaded procs      */
float    sv_gfactor0_out;       /*                   - swapped procs     */
float    sv_gfactor1_in;        /* Guaranteed residence factor 1         */
                                /*                   - loaded procs      */
float    sv_gfactor1_out;       /*                   - swapped procs     */
float    sv_ufactor_in;         /* University of Texas priority factor   */
float    sv_ufactor_out;        /* For interactive processes -           */
                                /*   Time since last interaction         */
                                /* For non-interactive processes -       */
                                /*   Time remaining                      */
                                /*     (cpu time limit - cpu time used)  */
int      sv_cpufactor;          /* # of running processes in-core to try */
                                /* for. Default is 8 + (2 * ncpu)        */
int      sv_bigproc;            /* If non-zero, processes above this     */
                                /* size in clicks won't be swapped unless*/
                                /* they're expanding or suspended.       */
word     sv_magic;              /* Magic number to indicate valid struct */
int      sv_size;               /* Size of schedvar structure.  This is  */
                                /* Used by the kernel to verify that the */
                                /* calling nschedv is in sync with the   */
                                /* kernel.                               */
int      sv_maxruns;            /* Maximum # of sched runs per second.   */
int      sv_smallproc;          /* Small interactive process click size  */
int      sv_itime;              /* Interaction time                      */
```

Values of 0 disable all scheduling variables except for `sv_max_outage`.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_ADMIN`	The process is allowed to use the `SVAR_SET` command.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_SYSPARAM permbit is allowed to use the SVAR_SET command.

RETURN VALUES

If schedv completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The schedv system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The user field length did not contain the schedvar structure to which *svartab* points.
EINVAL	A value in either the sv_magic or sv_size field did not match the value expected by the kernel.
EPERM	The SVAR_SET command was used and the process did not have appropriate privilege.

SEE ALSO

limit(2)

limit(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

nschedv(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`secstat`, `lsecstat`, `fsecstat` – Gets file security attributes

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secstat.h>
int secstat (char *path, struct secstat *buf);
int lsecstat (char *path, struct secstat *buf);
int fsecstat (int fildes, struct secstat *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `secstat` system call obtains the security attributes of a file; `fsecstat` system call obtains the same information for an open file. The `lsecstat` system call is similar to `secstat` except when the specified file is referenced by the link. In this case, `lsecstat` returns information about the link, and `secstat` returns information about the file referenced by the link.

The `secstat` and `lsecstat` system calls accept the following arguments:

path Specifies the file from which the security attributes are obtained.

buf Points to a `secstat` structure in which the information is returned.

The `fsecstat` system call accepts the following arguments:

fildes Specifies the file descriptor that identifies the file from which the security attributes are obtained.

buf Points to a `secstat` structure in which the information is returned.

A `secstat` structure includes the following members:

```
int      st_slevel;       /* File security level */
long     st_compart;      /* File compartments */
long     st_acldsk;       /* Access control disk address */
int      st_secflg;       /* Security flag */
int      st_intcls;       /* Class (not used) */
int      st_intcat;       /* Categories (not used) */
int      st_minlvl;       /* Device minimum security level */
int      st_maxlvl;       /* Device maximum security level */
long     st_valcmp;       /* Device authorized compartments */
```

NOTES

Any process may obtain the security attributes of a file labeled with a wildcard security label.

On a nondevice file, the `st_minlvl`, `st_maxlvl`, and `st_valcmp` fields of the `secstat` structure are equal to the minimum security level, maximum security level, and valid compartments, respectively, of the file system on which the file resides.

Only a process with appropriate privilege can retrieve the actual state of the file trap flags (`trapr` and `trapw`) in the `st_secflg` field.

The `secstat`, `lsecstat`, and `fsecstat` requests are not recorded in the security log.

The process must have read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component. (`secstat`/`lsecstat` systems call only.)

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to retrieve the trap state of the file.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list. (`secstat`/`lsecstat` system calls only.)
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label. (`secstat`/`lsecstat` system calls only.)
`PRIV_MAC_READ`	The process is granted read permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix (`secstat`/`lsecstat` system calls only) and is granted read permission to the file via the security label. The super user is allowed to retrieve the trap state of the file.

RETURN VALUES

If `secstat`, `lsecstat`, or `fsecstat` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `secstat` or `lsecstat` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`EACCES`	The *buf* argument points outside the process address space.

EACCES	The process is not granted read permission to the file via the security label, and the process does not have appropriate privilege.
EFAULT	The *path* argument points outside the process address space.
ENAMETOOLONG	The supplied file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
ESYSLV	The caller does not have an authorized `secadm` or `sysadm` category and is not a trusted process.

The `fsecstat` system call fails if one of the following error conditions occurs:

Error Code	**Description**
EBADF	The *fildes* argument is not a valid open file descriptor.
EBADF	The process is not granted read permission to the file via the security label, and the process does not have appropriate privilege.
EFAULT	The *buf* argument points outside the process address space.
EINVAL	The specified file is a socket.
ESYSLV	The caller does not have a authorized `secadm` or `sysadm` category, is not a trusted process, and is not authorized to execute at the security label of the file.

BUGS

When `secstat`, `lsecstat`, or `fsecstat` is used to obtain the labeling information for an inactive pty device special file, the labeling reported reflects the label and label range of the calling process. If the active label of the calling process is outside the label range of the calling process, the label and range returned reflects this. Since this is an illegal combination, any attempt to recreate a pty device node with these attributes fail. Because the pty device is automatically relabeled the next time it is used, this failure does not leave the pty device in an incorrectly labeled state even if it appears to do so.

FILES

`/usr/include/sys/secstat.h`	Defines secstat structure
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`getfacl`(2), `setdevs`(2), `setfacl`(2), `setfcmp`(2), `setfflg`(2), `setflvl`(2)

`spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`select` – Examines synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/param.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
```

`int select (int` *nfds*`, fd_set *`*readfds*`, fd_set *`*writefds*`, fd_set *`*exceptfds*`,`
`struct timeval *`*timeout*`);`

`FD_SET (`*fd*`, `*&fdset*`);`
`FD_CLR (`*fd*`, `*&fdset*`);`
`FD_ISSET (`*fd*`, `*&fdset*`);`
`FD_ZERO (`*&fdset*`);`
`int` *fd*`;`
`fd_set` *fdset*`;`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `select` system call examines I/O descriptor sets to determine whether the files associated with the specified file descriptors are ready for reading, are ready for writing, or have an exceptional condition pending.

An I/O descriptor set is an array of long integers where each bit in the array corresponds to a file descriptor defined for the process. The leftmost bit in the first array element corresponds to file descriptor 0, and so on. The number of bits allocated in each descriptor set is defined by the parameter `FD_SETSIZE` in header file `<sys/types.h>` which corresponds to the maximum number of files a process can have open concurrently.

The `select` system call accepts the following arguments:

nfds Specifies the number of file descriptors that are to be checked in the descriptor sets pointed to by *readfds*, *writefds*, and *exceptfds*. The bits from 0 through *nfds* -1 in the descriptor sets are examined.

readfds Points to an I/O descriptor set used to specify which file descriptors are to be examined to determine if the associated files are ready for reading. If no file descriptors are to be examined for reading, specify a null pointer.

writefds Points to an I/O descriptor set used to specify which file descriptors are to be examined to determine if the associated files are ready for writing. If no file descriptors are to be examined for writing, specify a null pointer.

exceptfds Points to an I/O descriptor set used to specify which file descriptors are to be examined to determine if the associated files have any exceptional conditions pending. If no file descriptors are to be examined for exceptional conditions, specify a null pointer.

timeout Points to a `timeval` structure which specifies the maximum interval to wait for the examination to complete. If *timeout* is a null pointer, `select` is blocked indefinitely. To cause a poll, the *timeout* argument must be nonzero, pointing to a `timeval` structure containing 0 values.

The `select` system call returns, in place, descriptor sets of the file descriptors that are ready. The value returned by `select` is the total number of file descriptors which are ready.

The following macros are provided for manipulating I/O descriptor sets. In these descriptions, *fd* stands for file descriptor, and *fdset* refers to file descriptor sets.

`FD_CLR` (*fd*, *&fdset*)	Removes *fd* from *fdset*.
`FD_ISSET` (*fd*, *&fdset*)	Returns nonzero if *fd* is a member of *fdset*; otherwise, returns 0.
`FD_SET` (*fd*, *&fdset*)	Includes a particular in *fd* in *&fdset*.
`FD_ZERO` (*&fdset*)	Sets all bits in *fdset* to 0.

The behavior of these macros is undefined if a descriptor value is less than 0 or greater than or equal to `FD_SETSIZE`, which is normally equal to the maximum number of files a process can have open concurrently. A program may give `FD_SETSIZE` a larger value by defining it before the inclusion of header file `<sys/types.h>`.

NOTES

The active security label of the calling process must be greater than or equal to the security label of each file. If this condition is not met for a given file descriptor, events on that file descriptor are ignored.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_READ`	The calling process is allowed to override the security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security label restrictions.

RETURN VALUES

If `select` completes successfully, it returns the number of descriptors contained in the descriptor sets. If the time limit expires, `select` returns 0.

Because failure to meet the security label requirements causes a file descriptor to be ignored, this failure alone does not result in an error.

If an error occurs, a value of –1 is returned, the descriptor sets remain unmodified (even in the case of an interrupted call), and `errno` is set to indicate the error.

ERRORS

An error return from the `select` system call indicates one of the following conditions:

Error Code	Description
EBADF	One of the descriptor sets specified a descriptor that was not valid.
EFAULT	The argument address is not valid.
EINTR	A signal was delivered before any of the selected events occurred, or the time limit expired.
EINVAL	The specified time limit is not valid; one of its components is negative or too large, or *nfds* is less than or equal to 0.

BUGS

The current implementation works only for the master side of a pty, a tty, a pipe, and sockets.

The `select` system call should probably return the time remaining from the original time-out, if any, by modifying the time value in place. This may be implemented in future versions of the system. Therefore, do not assume that the time-out value will be unmodified by `select`.

EXAMPLES

The following example shows how to use the `select` system call:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

/*
 *
 *
 *      Returns:
 *              0, if read would block
 *              1, if read would not block
 */
chkread(fd)
int fd;                                 /* File descriptor */
{
        int nfound;                     /* Number of ready descriptors */
        fd_set readfds;                 /* Read file descriptors bit mask */
        struct timeval timeout;


        FD_ZERO(&readfds); FD_SET(fd, &readfds);

        timeout.tv_sec = 0;             /* Cause select to return immediately */
        timeout.tv_usec = 0;

        while ((nfound = select(FD_SETSIZE, &readfds, 0, 0, &timeout)) == -1) {
               if (errno == EINTR)
                     continue;         /* Ignore interrupts */

                fprintf(stderr, "select() failed, errno = %d\n", errno);
                exit(1);
        }
        return(nfound);
 }
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `select` system call

SEE ALSO

`read`(2), `write`(2)

NAME

`semctl` – Provides semaphore control operations

SYNOPSIS

```
#include <sys/sem.h>

int semctl (int semid, int semnum, int cmd, union semun arg...);

union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
};
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `semctl` system call provides a variety of semaphore control operations as specified by *cmd*. It accepts the following arguments:

semid Specifies a semaphore identifier associated with a set of semaphores.

semnum Identifies the semaphore in the *semid* group.

cmd Specifies a semaphore control operation. The following are valid *cmd* values.

The following semaphore control operations are executed for the semaphore specified by *semid* and *semnum*. The level of permission required for each operation is specified with each command (see `sem`(5) `ipc`(7)).

`GETVAL` Returns the value of `semval` (see `sem`(5)). This command requires read permission.

`SETVAL` Sets the value of `semval` to *arg*.`val`. When this command is successfully executed, the `semadj` value corresponding to the specified semaphore in all processes is cleared. This command requires alter permission.

`GETPID` Returns the value of `sempid`. This command requires read permission.

`GETNCNT` Returns the value of `semncnt`. This command requires read permission.

`GETZCNT` Returns the value of `semzcnt`. This command requires read permission.

The following values for *cmd* operate on each `semval` in the set of semaphores (see `sem`(5)):

GETALL Places `semval`s into the array (of type `unsigned short`) pointed to by *arg*.`array`. This command requires read permission.

SETALL Sets `semval`s according to the array (of type `unsigned short`) pointed to by *arg*.`array`. When this *cmd* is successfully executed, the `semadj` values corresponding to each specified semaphore in all processes are cleared. This command requires alter permission.

The following values for *cmd* are also available (see `ipc`(5)):

IPC_STAT Places the current value of each member of the `semid_ds` data structure associated with *semid* into the `semid_ds` structure pointed to by *arg*.`buf`. The contents of this structure are defined in `sem`(5). This command requires read permission.

IPC_SET Sets the value of the members of the `semid_ds` data structure associated with *semid* to the corresponding value found in the `semid_ds` structure pointed to by *arg*.`buf`. See the following:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode     /* only access permission bits */
```

The mode bits specified in `ipc`(7) are copied into the corresponding bits of the `sem_perm.mode` associated with *semid*. The values of any other bits are unaltered.

The `IPC_SET` command can be executed only by a process that has an effective user ID equal to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` data structure associated with *semid*.

IPC_RMID Removes the semaphore identifier specified by *semid* from the system and destroys the set of semaphores and `semid_ds` data structure associated with it. This command can be executed only by a process that has an effective user ID equal to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` data structure associated with *semid*.

IPC_SETACL Sets the access control list (ACL) on the semaphore set specified by *semid*. The `ipc_perm` structure within the `semid_ds` structure pointed to by *buf* contains a pointer, `ipc_acl`, to an `acl_rec` structure with the required ACL entries, and a count of those entries, `ipc_aclcount`. If an ACL exists for the semaphore set, it is replaced by the one provided with this call. If `ipc_aclcount` is 0, any existing ACL is removed. The calling process must be the owner of the semaphore set specified by *semid*.

`IPC_GETACL` Retrieves the access control list (ACL) for the semaphore set specified by *semid*. The `ipc_perm` structure within the `semid_ds` structure pointed to by *buf* contains a pointer, `ipc_acl`, to an `acl_rec` structure where the ACL entries are to be returned. The count of entries to be returned is specified in the `ipc_aclcount` field. If there are more than `ipc_aclcount` entries, only the first `ipc_aclcount` is returned. If there are fewer than `ipc_aclcount` entries, all entries are returned. The return value indicates the number of entries returned. If there is no ACL, the return value is 0. The calling process must have read permission to the semaphore set specified by *semid*.

`IPC_SETLABEL`
Sets the security label on the semaphore set specified by *semid*. The `ipc_perm` structure within the `semid_ds` structure pointed to by *buf* contains a security level, `ipc_slevel`, and a compartment set, `ipc_scomps`, to be set in the security label on the semaphore set. Only a process with the appropriate privilege can perform this operation.

arg Specifies an optional structure used by the *cmd* argument.

NOTES

A process is granted read permission to a semaphore set only if the active security label of the process is greater than or equal to the security label of the semaphore set, and the process is granted read access by the semaphore set ACL (if one is assigned). This applies to the `IPC_STAT` and `IPC_GETACL` operations.

The `IPC_SET`, `IPC_RMID`, and `IPC_SETACL` operations require that the active security label of the process is equal to the security label of the semaphore set.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_READ`	The process is considered to meet the security label requirements for being granted read permission to a semaphore set.
`PRIV_MAC_WRITE`	The process is considered to meet the security label requirements for performing an `IPC_SET`, `IPC_RMID`, or `IPC_SETACL` operation.
`PRIV_DAC_OVERRIDE`	The process is considered to meet the permission mode and ACL requirements for being granted read permission to a semaphore set.
`PRIV_FOWNER`	The process is considered to meet the semaphore set ownership requirements for the `IPC_SET`, `IPC_RMID`, and `IPC_SETACL` operations.

If the `PRIV_SU` configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above.

The super user is considered the owner of a semaphore set, and is granted read permission to that semaphore set.

RETURN VALUES

Upon successful completion, the value returned by `semctl` depends on *cmd*, as follows:

`GETVAL`	Value of *semval*
`GETPID`	Value of *sempid*
`GETNCNT`	Value of *semncnt*
`GETZCNT`	Value of *semzcnt*
All others	Value of 0

Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `semctl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Operation permission is denied to the calling process (see `sem`(5)).
`EACCES`	The *cmd* argument is `IPC_GETACL`, and the calling process does not have read permission.
`EFAULT`	*arg.*`buf` points to an illegal address.
`EFAULT`	The *cmd* argument is `IPC_SETACL` or `IPC_GETACL`, and the *ipc_acl* field in *buf* points to an illegal address.
`EINVAL`	The *semid* argument is not a valid semaphore identifier.
`EINVAL`	The *semnum* argument is less than 0 or greater than (`sem_nsems` –1).
`EINVAL`	The *cmd* argument is not a valid command.
`EINVAL`	The *cmd* argument is `IPC_SET`, and `sem_perm.uid` or `sem_perm.gid` is not valid.

`EINVAL` The *cmd* argument is `IPC_SETACL`, and one of the following is true:

- The `ipc_aclcount` field in *buf* is 0, but there is no ACL associated with *msqid*.
- The `ipc_aclcount` field in *buf* is less than 0 or greater than 256.
- The ACL supplied failed validation.

`ENOMEM` The *cmd* argument is `IPC_SETACL`, and no memory was available to store the ACL. The command should be retried at a later time.

`EPERM` The *cmd* argument is equal to `IPC_RMID` or `IPC_SET`, and the effective user ID of the calling process is not equal to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` data structure associated with *semid*, and the calling process does not have appropriate privilege.

`EPERM` The *cmd* argument is `IPC_SETLABEL`, and the calling process does not have appropriate privilege.

`EPERM` The *cmd* argument is `IPC_SETACL`, and the calling process does not meet ownership requirements and does not have appropriate privilege.

`ERANGE` The *cmd* argument is `SETVAL` or `SETALL`, and the value to which `semval` is to be set is greater than the system-imposed maximum.

FILES

`/usr/include/sys/sem.h` Contains semaphore-related data structures and macros

SEE ALSO

`semget`(2), `semop`(2)

`ipcs`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ipc`(5), `sem`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`semget` – Provides access to semaphore identifiers

SYNOPSIS

```
#include <sys/sem.h>
int semget (key_t key, int nsems, int semflg);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The `semget` system call returns the semaphore identifier associated with *key*. It accepts the following arguments:

key Specifies the semaphore.

nsems Specifies the number of semaphores to allocate for the *key*.

semflg Specifies a flag value.

A semaphore identifier, with its associated `semid_ds` data structure and set containing *nsems* semaphores (see `sem`(5)), is created for *key* if one of the following is true:

- *key* is equal to `IPC_PRIVATE`.
- *key* does not already have a semaphore identifier associated with it, and *semflg*&`IPC_CREAT` is not 0.

Upon creation, the `semid_ds` data structure associated with the new semaphore identifier is initialized as follows:

- `sem_perm.cuid`, `sem_perm.uid`, `sem_perm.cgid`, and `sem_perm.gid` are set to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of `sem_perm.mode` are set to the low-order 9 bits of *semflg*.
- `sem_nsems` is set to the value of *nsems*.
- `sem_otime` is set to 0, and `sem_ctime` is set to the current time.
- The data structure associated with each semaphore in the set is not initialized. The `SETVAL` or `SETALL` command of the `semctl`(2) system call can be used to initialize each semaphore.

NOTES

If the calling process has the `ipc_persist` permission bit, the semaphore set will be created as a persistent set. Persistent semaphore sets will not be removed from the system unless a `semctl`(2) system call with the command `IPC_RMID` or an `ipcrm`(1) command is performed on the set.

If the calling process does not have this permission bit, the semaphore set will be linked into a list of nonpersistent sets belonging to the session of which the process is a member. When the last process of the session terminates, all the semaphore sets linked to the session will be removed from the system.

A process with the effective privilege shown is granted the following abilities:

Privilege	**Description**
`PRIV_RESOURCE`	The process is considered to have the `ipc_persist` permission bit.

If the `PRIV_SU` configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is considered to have the `ipc_persist` permission bit.

RETURN VALUES

If `semget` completes successfully, a nonnegative integer, namely a semaphore identifier, is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `semget` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted (see `ipc`(7)).
`EEXIST`	A semaphore identifier exists for *key* but both *semflg*&`IPC_CREAT` and *semflg*&`IPC_EXCL` are not 0.
`EINVAL`	The value of *nsems* is either less than or equal to 0 or greater than the system-imposed limit.
`EINVAL`	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to 0.
`ENOENT`	A semaphore identifier does not exist for *key*, and *semflg*&`IPC_CREAT` is 0.
`ENOSPC`	A semaphore identifier is to be created, but the system-imposed limit on the maximum number of allowed semaphore identifiers system-wide would be exceeded.

FILES

`/usr/include/sys/sem.h` Contains semaphore-related data structures and macros

SEE ALSO

`semctl`(2), `semop`(2)

`ipcrm`(1), `ipcs`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`stdipc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`ipc`(5), `sem`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

semop – Provides general semaphore operations

SYNOPSIS

```
#include <sys/sem.h>
int semop (int semid, struct sembuf *sops, size_t nsops);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

XPG4

DESCRIPTION

The semop system call is used to perform an array of semaphore operations atomically on the set of semaphores associated with the semaphore identifier.

The semop system call accepts the following arguments:

semid Specifies a semaphore identifier associated with a set of semaphores.

sops Points to the array of semaphore-operation structures.

nsops Specifies the number of such structures in the array. Each sembuf structure includes the following members:

```
short sem_num;    /* semaphore number */
short sem_op;     /* semaphore operation */
short sem_flg;    /* operation flags */
```

Each semaphore operation specified by sem_op is performed on the corresponding semaphore specified by *semid* and sem_num. See the sem(5) man page for information on the available types of permissions. The variable sem_op specifies one of three semaphore operations, as follows:

1. If sem_op is a negative integer and the calling process has alter permission, one of the following actions occurs:
   - If semval (see sem(5)) is greater than or equal to the absolute value of sem_op, the absolute value of sem_op is subtracted from semval. Also, if *sem_flg*&SEM_UNDO is not 0, the absolute value of sem_op is added to the calling process' semadj value for the specified semaphore (see exit(2)).
   - If semval is less than the absolute value of sem_op and *sem_flg*&IPC_NOWAIT is not 0, semop returns immediately.

- If `semval` is less than the absolute value of `sem_op` and *sem_flg*&`IPC_NOWAIT` is 0, `semop` increments the `semncnt` associated with the specified semaphore and suspends execution of the calling process until one of the following conditions occurs:
  - `semval` becomes greater than or equal to the absolute value of `sem_op`. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, the absolute value of `sem_op` is subtracted from `semval`, and, if *sem_flg*&`SEM_UNDO` is not 0, the absolute value of `sem_op` is added to the calling process' `semadj` value for the specified semaphore.
  - The *semid* for which the calling process is awaiting action is removed from the system (see `semctl`(2)). When this occurs, `errno` is set equal to `EIDRM`, and a value of –1 is returned.
  - The calling process receives a signal that is to be caught. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the `sigaction`(2) system call.

2. If `sem_op` is a positive integer and the calling process has alter permission, the value of `sem_op` is added to `semval`, and, if *sem_flg*&`SEM_UNDO` is true, the value of `sem_op` is subtracted from the calling process's `semadj` value for the specified semaphore.
3. If `sem_op` is 0 and the calling process has read permission, one of the following actions occurs:
   - If `semval` is 0, `semop` returns immediately.
   - If `semval` is not equal to 0 and *sem_flg*&`IPC_NOWAIT` is not 0, `semop` returns immediately.
   - If `semval` is not equal to 0 and *sem_flg*&`IPC_NOWAIT` is also 0, `semop` increments the `semzcnt` associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
     - `semval` becomes 0, at which time the value of `semzcnt` associated with the specified semaphore is decremented.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, `errno` is set equal to `EIDRM`, and a value of –1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of `semzcnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the `sigaction`(2) system call.

Upon successful completion, the value of `sempid` for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

NOTES

A process is granted read permission to a semaphore set only if the active security label of the process is greater than or equal to the security label of the semaphore set, and the process is granted read access by the semaphore set access control list (ACL) (if one is assigned).

A process is granted write permission to a semaphore set only if the active security label of the process is equal to the security label of the semaphore set, and the process is granted write access by the semaphore set ACL (if one is assigned).

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_MAC_READ	The process is considered to meet the security label requirements for being granted read permission to a semaphore set.
PRIV_MAC_WRITE	The process is considered to meet the security label requirements for being granted write permission to a semaphore set.
PRIV_DAC_OVERRIDE	The process is considered to meet the permission mode and ACL requirements for being granted read and write permission to a semaphore set.

If the PRIV_SU configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is granted read and write permission to a semaphore set.

RETURN VALUES

If semop completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The semop system call fails if one or more of the following are true for any of the semaphore operations specified by *sops*:

Error Code	Description
E2BIG	The *nsops* argument is greater than the system-imposed maximum.
EACCES	Operation permission is denied to the calling process (see ipc(7)).
EAGAIN	The operation would result in suspension of the calling process, but *sem_flg*&IPC_NOWAIT is not 0.
EFAULT	The *sops* argument points to an illegal address.
EFBIG	The *sem_num* field is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.
EIDRM	The semaphore identifier *semid* was removed from the system.
EINTR	The semop system call was interrupted by a signal.
EINVAL	The *semid* argument is not a valid semaphore identifier.
EINVAL	The number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the limit.

`ENOSPC`	The limit on the number of individual processes requesting a `SEM_UNDO` would be exceeded.
`ERANGE`	An operation would cause a `semval` value to overflow the system-imposed limit.
`ERANGE`	An operation would cause a `semadj` value to overflow the system-imposed limit.

FILES

`/usr/include/sys/sem.h`	Contains semaphore-related data structures and macros

SEE ALSO

`exec`(2), `exit`(2), `fork`(2), `semctl`(2), `semget`(2), `sigaction`(2)

`ipcs`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ipc`(5), `sem`(5), `types`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`send`, `sendmsg`, `sendto` – Sends a message from a socket

SYNOPSIS

All Cray Research systems:

```
#include <sys/types.h>
#include <sys/socket.h>

int send (int s, char *buf, int len, int flags);

int sendto (int s, char *buf, int len, int flags, struct sockaddr *to,
int tolen);
```

Cray PVP systems:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendmsg (int s, struct msghdr *buf, int flags);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `send`, `sendmsg`, and `sendto` system calls transmit a message (*buf*) to another socket.

You can use a `send` call only on a connected socket. You can use `sendto` or `sendmsg` on either a connected or unconnected socket. The `sendmsg` system call uses the same `msghdr` structure as the `recvmsg`(2) system call to minimize the number of directly supplied arguments. For more information, see the `connect`(2) and `recv`(2) man pages.

The `send`, `sendmsg`, and `sendto` system calls accept the following arguments:

s Specifies the descriptor for a socket.

buf Points to the address of the message to be sent. See `recv`(2) for a description of the `msghdr` structure.

len Specifies the number of bytes to be sent. If the message is too long to pass atomically through the underlying protocol, the error message `EMSGSIZE` is returned, and the message is not transmitted.

flags Specifies optional flags that control transmission of the message. The following are valid values for *flag*:

`MSG_OOB` Specifies that the message should be sent out-of-band on sockets that support such a notion. Out-of-band messages correspond to the TCP notion of urgent data.

MSG_DONTROUTE	Specifies that the message be sent without using local routing tables. Allows the caller to take control of routing (for example, in network debugging software).

to Points to a sockaddr structure that must be filled with the destination address.

tolen Specifies the length of the destination address, specified by *to*.

The send system call is unreliable. A return value of −1 indicates some locally detected errors, but you cannot determine whether the message was received. The send call only queues data for transmission. When send returns 0, it indicates that the message was put in the queue. If the message is not received, error messages are unavailable.

If no message space is available at the socket to hold the message to be transmitted, send usually waits for space to become available, unless the socket was placed in the nonblocking I/O mode by an ioctl(2) request of FIONBIO. You can use the select(2) call to determine when it is possible to send more data.

NOTES

These system calls can be subjected to additional security rules. The two sockets being connected each have security attributes that are inherited from their associated processes. These attributes must be equal if the SOCKET_MAC option is enabled. In addition, the network and remote host have security-attribute ranges, which are specified in the network access list (NAL) portion of the spnet.conf configuration file and administered with the spnet(8) command.

If the SOCKET_MAC option is not enabled, the security attributes of the socket are not required to be equal, but the security range of the process, which is specified in the UDB for the user, must include the minimum label for the remote host as specified in the NAL. Note that SOCKET_MAC is part of TCP/IP configurable feature variables list in uts/cf/Nmakefile.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_MAC_WRITE	The process is allowed to override the security label restrictions when the SOCKET_MAC option is enabled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override security label restrictions when the SOCKET_MAC option is enabled.

RETURN VALUES

If send, sendmsg, or sendto completes successfully, the number of characters sent is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The `send`, `sendmsg`, or `sendto` system call fails if one of the following conditions occurs:

Error Code	Description
EACCES	Permission is denied (because of a security violation).
EACCES	If the SOCKET_MAC option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EBADF	Descriptor *s* is not valid.
EFAULT	Invalid user space address is specified for a parameter.
EMSGSIZE	Socket requires the message to be sent atomically, and the size of the message to be sent makes this impossible.
EMSGSIZE	The `msg_iovlen` field is greater than or equal to the MSG_MAXIOVLEN parameter (defined in the `sys/socket.h` file).
ENOBUFS	System cannot allocate an internal buffer. The operation can succeed when buffers become available.
ENOBUFS	Output queue for a network interface is full. This generally indicates that the interface has stopped sending, but it can indicate transient congestion.
ENOTSOCK	Descriptor *s* is not a socket.
EWOULDBLOCK	Socket is marked nonblocking, and the requested operation would block.

FILES

`/etc/config/spnet.conf`	Network access list file
`/usr/adm/sl/slogfile`	Receives security log records
`/usr/include/sys/socket.h`	Contains definitions related to sockets, types, address families, and options
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`ioctl`(2), `recv`(2), `select`(2), `socket`(2)

`slog`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`spnet`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

NAME

`setash` – Sets an array session handle

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

#int setash (ash_t ash);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The `setash` system call changes the handle for the array session containing the current process to the specified value. The current process must have super-user privileges to invoke the `setash` system call.

Ordinarily, a handle that is unique within the current system is assigned to an array session when the array session is created with the `newarraysess`(2) system call. The `setash` system call can override this default handle, perhaps for assigning a handle that is unique across an entire array or for synchronizing handles with an array session on another system.

The `setash` system call accepts the following argument:

ash Represents the array session handle that is to be assigned to the current array session. The handle specified by *ash* must be a positive value, must not be in use on the current system, and must not be in the range of values that UNICOS uses for default array session handles. The range of default handles is defined by the system variables `minash` and `maxash`.

RETURN VALUES

If `setash` completes successfully, a value of 0 is returned; otherwise, a value of `-1` is returned, and `errno` is set to indicate the error.

ERRORS

The `setash` system call fails if one of the following conditions occurs:

Error Code	**Description**
`EINVAL`	*ash* is negative, in use by another array session on this system, or in the range of values reserved by the system for default array session handles.
`EPERM`	The current process does not have super-user privileges.

SEE ALSO

getash(2), newarraysess(2)

array_services(7), array_sessions(7)

NAME

`setdevs` – Sets file security label and security flag attributes

SYNOPSIS

```
#include <sys/secdev.h>
int setdevs (char *dname, struct secdev *sdev);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setdevs` system call sets the minimum, maximum, and active security labels, and the security flags of a file to the values contained in the `secdev` structure.

The `setdevs` system call accepts the following arguments:

dname Specifies the file for which the security labels and flags are set.

sdev Points to the `secdev` structure which contains the security values to be set.

A `secdev` structure includes the following members:

```
int     dv_minlvl;      /* minimum security level */
int     dv_maxlvl;      /* maximum security level */
int     dv_actlvl;      /* active security level */
long    dv_valcmp;      /* authorized compartments */
long    dv_actcmp;      /* active compartments */
int     dv_intcls;      /* active integrity class (not used) */
long    dv_intcat;      /* active integrity categories (not used) */
int     dv_devflg;      /* device security flags */
int     dv_devprv;      /* device privileges */
```

NOTES

Only an appropriately privileged process can set the minimum or maximum security label of a device special file. Any process can attempt to set the minimum or maximum security label of a file that is not a device special file. If the file is not a device special file, then the supplied minimum and maximum security labels are ignored, and the file's minimum and maximum security labels are set to the minimum and maximum security labels of the file system on which the file resides.

Only an appropriately privileged process can downgrade the active security label of the specified file. If the specified file is an empty directory, then any process can upgrade the active security label of the file. Otherwise, only an appropriately privileged process can upgrade the active security label of the file.

If the specified file is a device special file, and the supplied security flags include the `mldev` flag, then the supplied active security label is ignored, and the active security label of the file is set to the supplied maximum security label.

If the supplied file is a device special file, and the supplied security flags do not include the `secdv` flag, then the `secdv` and `state` flags are turned off for the specified file.

This system call changes only the state of the `state`, `secdv`, `mldev`, and `entry` security flags. No other security flags are changed.

If the file is not a device special file, attempts to enable `state` are ignored. If the file is a device special file, attempts to enable `state` without also enabling `secdv` are ignored. Disabling `secdv` automatically causes `state` to be disabled.

Attempts to change the security label or flags on a public device are ignored. Attempts to change the flags on a pseudo tty are ignored.

A user is allowed to upgrade the label on his/her directory if the directory is empty and the user has write access to the directory.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_ADMIN`	The process is allowed to set the minimum and maximum security label of the file.
`PRIV_ADMIN`	The process is allowed to set the security flags of the file.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
`PRIV_MAC_DOWNGRADE`	The process is allowed to downgrade the active security label of the file.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label.
`PRIV_MAC_UPGRADE`	The process is allowed to upgrade the active security label of the file.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix. The super user is allowed to set the minimum, maximum, and active security label of the file, and the security flags of the file.

RETURN VALUES

If `setdevs` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setdevs` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`ECOMPV`	The specified compartments are not valid within the compartments of the system.
`ECOMPV`	If the `MLS_OBJ_RANGES` configuration option is enabled, and the specified compartments are not valid compartments for the file.
`EFAULT`	Error occurred in reading `setdev` structure.
`EINVAL`	The security label parameters are not valid.
`EINVFS`	The file system on which the file exists is a pre-UNICOS 6.0 file system.
`ENAMETOOLONG`	The *dname* argument is longer than allowed by `PATH_MAX`.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have permission to upgrade an empty directory.
`ESECADM`	The process does not have appropriate privilege to use this system call.
`ESYSLV`	If the `MLS_OBJ_RANGES` configuration option is enabled, and the specified levels are not within the security level range of the system.

SEE ALSO

`secstat`(2), `setfacl`(2), `setfcmp`(2), `setfflg`(2)

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`spdev`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setfacl` – Sets access control list for file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/acl.h>

int setfacl (char *fname, struct acl *aclents, int count);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setfacl` system call sets the access control list (ACL) of a file to the value specified in an array. A file's ACL controls, in part, access to the file. Only the file owner or a process with appropriate privilege can set an ACL on a file. If the process is not a member of the owning group of the file, the set-group-ID mode bit of the file is cleared unless the process has appropriate privilege. If the `FSETID_RESTRICT` system configuration parameter is enabled, the set-user-ID and set-group-ID mode bits of a file are cleared unless the process has appropriate privilege.

The `setfacl` system call accepts the following arguments:

fname Specifies the file for which the ACL is set.

aclents Specifies an array of ACL entries.

count Indicates the number of entries in the array; cannot exceed 256.

NOTES

A `setfacl` request replaces any previously existing ACL on the file.

If the system is configured with discretionary access violation logging enabled, all errors are recorded in the security log (except errors of type `ENOENT`, `ENOTDIR`, and `ENAMETOOLONG`).

The process must have write permission to the file via the security label. That is, the active security label of the process must be equal to the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to the component via the permission bits and ACL.
`PRIV_FOWNER`	The process is considered the owner of the file.

`PRIV_FSETID`	The file's set-user-ID or set-group-ID mode bit are not cleared.
`PRIV_MAC_READ`	The process is granted search permission to the component via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to a component of the path prefix via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the file via the security label. The super user is considered the file owner and is allowed to set an ACL on a file whose mode includes the set-user-ID or set-group-ID mode bit. If the caller is the super user, the file's set-user-ID and set-group-ID mode bits are not cleared.

RETURN VALUES

If `setfacl` completes successfully, the number of elements in the file's ACL is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setfacl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`EFAULT`	The *fname* argument points outside the process address space.
`EFAULT`	The *aclents* argument points outside the process address space.
`EINVAL`	The *count* argument is less than 0. If *count* exceeds 256, its value is truncated to 256.
`EINVAL`	The specified file resides on a nonnative file system.
`EMANDV`	The process does not have write permission to the file via the security label.
`ENAMETOOLONG`	The specified file name is too long.
`ENOACL`	The file's previously assigned ACL, if one exists, is corrupted.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EOWNV`	The process is not the file owner and does not have appropriate privilege.

EXAMPLES

This example shows how to use the `setfacl` system call to create ACL entries for a file.

First, a `getfacl`(2) system call displays (on `stdout`) the current ACL entries for the file, `argv[1]`. After this display, the program allows the user to add entries to the ACL. A `setfacl` request then creates a new ACL for the designated file.

`ACLSIZE`, defined in the `sys/acl.h` file, specifies the maximum number of entries that can exist in an ACL.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/acl.h>
#include <pwd.h>

main(int argc, char *argv[])
{
     struct acl buf[ACLSIZE];
     struct passwd *pwptr;
     char lid[15];
     int no, i;

     if ((no = getfacl(argv[1], buf, ACLSIZE)) == -1) {
          perror("getfacl failed");
          exit(1);
     }

     printf("Access control list for %s currently contains", argv[1]);
     printf(" the following users:\n\n");
     printf("ID         Login ID   Name");
     printf("                       Permissions\n\n");

     for (i = 0; i < no; i++) {
          pwptr = getpwuid(buf[i].ac_usid);
          printf("%-5d      %-10s %-25s   %c%c%c\n",
                  buf[i].ac_usid, pwptr->pw_name, pwptr->pw_gecos,
                  buf[i].ac_mode & 04 ? 'r' : ' ',
                  buf[i].ac_mode & 02 ? 'w' : ' ',
                  buf[i].ac_mode & 01 ? 'x' : ' ');
     }

     /* Add entries to access control list. */
     printf("\nWhich entries are to be added (q to quit)?\n\n");
     while(1) {
          printf("User's login ID --> ");
          gets(lid);
          if (strcmp(lid, "q") == 0) break;
          if ((pwptr = getpwnam(lid)) == NULL) {
               fprintf(stderr, "Invalid login ID!\n");
               continue;
          }
          buf[no].ac_usid = pwptr->pw_uid;
          buf[no].ac_grid = pwptr->pw_gid;
```

```
                buf[no].ac_flag = FLAG_UIDGID;
                buf[no].ac_mode = 04;  /* allow read permission only */
                buf[no].ac_sort = 0;
                buf[no].ac_same = 0;
                no++;                 /* increment number of ACL entries */
           }

           if (setfacl(argv[1], buf, no) == -1) {
                perror("setfacl failed");
                exit(1);
           }
      }
```

SEE ALSO

`getfacl`(2), `rmfacl`(2), `secstat`(2), `setdevs`(2), `setfcmp`(2), `setfflg`(2), `setflvl`(2)

`spacl`(1), `spclr`(1), `spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setfcmp` – Sets file compartments

SYNOPSIS

`#include <unistd.h>`

`int setfcmp (char *`*fname*`, long` *fcmp*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setfcmp` system call sets the compartments of a file to the value specified by the compartment bit mask. A file's compartments control, in part, access to the file on a UNICOS system. Only a process with appropriate privilege can use this system call.

The `setfcmp` system call accepts the following arguments:

fname Specifies the file for which the compartments are set.

fcmp Specifies the compartment bit mask which determines the value of the compartments to be set.

NOTES

All `setfcmp` requests are recorded in the security log, indicating success or failure (except errors of type `ENOENT`, `ENOTDIR`, and `ENAMETOOLONG`).

A user is allowed to upgrade the label on his or her directory if the directory is empty and the user has write access to the directory.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
`PRIV_MAC_DOWNGRADE`	The process as allowed to use this system call to set the file's compartments to a value that does not include every compartment that is currently associated with the file.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label.

`PRIV_MAC_UPGRADE`	The process is allowed to use this system call to set the file's compartments to a value that includes at least every compartment that is currently associated with the file.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is allowed to upgrade or downgrade the file's compartments.

RETURN VALUES

If `setfcmp` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setfcmp` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`ECOMPV`	The requested compartments are not authorized for the file system in which the file resides.
`EFAULT`	The *fname* argument points outside the process address space.
`EINVAL`	The specified file resides on a nonnative file system.
`ENAMETOOLONG`	The specified file name is too long.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have permission to upgrade an empty directory.
`ESECADM`	The process does not have appropriate privilege to use this system call.

The security administrator's security level range must include the security level of the file, and the security administrator's authorized compartments must dominate the specified compartments of the file.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setfcmp` system call

SEE ALSO

`secstat`(2), `setdevs`(2), `setfacl`(2), `setfflg`(2), `setflvl`(2)

`spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

setfflg – Sets file security flags

SYNOPSIS

```
#include <unistd.h>
#include <sys/tfm.h>

int setfflg (char *fname, long flags);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setfflg` system call sets the security flags of a file to the value specified by flag bit mask. A file's security flags indicate whether a file requires special handling. Only a process with appropriate privilege can use this system call.

The `setfflg` system call accepts the following arguments:

fname Specifies the file for which the flags are set.

flags Specifies the flag bit mask which determines the value of the flags to be set.

NOTES

Only the `ml_symlink`, `exec`, `trapr`, `trapw`, `mldev`, and `entry` flags can be set using the `setfflg` system call. Requests to set other flags are ignored. The `secdv` flag can be cleared but not set by `setfflg`. The `exec` flag is obsolete on UNICOS 9.1 and later systems.

If the specified file is a device special file, and the supplied security flags include the `mldev` flag, the active security label of the file is sent to the maximum security label.

The `ml_symlink` flag is the only flag that can be placed on a symbolic link. Attempts to set `ml_symlink` and additional flags on a symbolic link in a single request results in an error. Attempts to set the `mld_symlink` flag on anything other than a symbolic link results in an error. Attempts to change the flags on a public device or a pseudo ttyp are ignored.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to use this system call.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list.

PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is allowed to use this system call.

RETURN VALUES

If setfflg completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setfflg system call fails if one of following error conditions occurs:

Error Code	**Description**
EACCES	A component of the path prefix denies search permission.
EFAULT	The *fname* argument points outside the process address space.
EINVAL	The specified file resides on a nonnative file system.
EINVFS	The file system on which the file exists is from a release previous to the UNICOS 6.0 release.
ENAMETOOLONG	The specified file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
ESECADM	The process does not have appropriate privilege to use this system call.
ESECFLGV	Requested security flags are not authorized for the UNICOS system.
ESECFLGV	Requested security flags are not allowed for the object type.

FILES

/usr/include/sys/tfm.h	
/usr/include/unistd.h	Contains C prototype for the setfflg system call

SEE ALSO

secstat(2), setdevs(2), setfacl(2), setfcmp(2), setflvl(2)

spset(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

slog(4), slrec(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setflvl` – Sets security level of a file

SYNOPSIS

```
#include <unistd.h>
int setflvl (char *fname, int flvl);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setflvl` system call sets the security level of file to a specified value. A file's security level controls, in part, access to the file on a UNICOS system. Only a process with appropriate privilege can use this system call.

The `setflvl` system call accepts the following arguments:

fname Specifies the file for which the level is set.

flvl Specifies the security level.

NOTES

All `setflvl` requests are recorded in the security log, indicating success or failure (except errors of type `ENOENT`, `ENOTDIR`, and `ENAMETOOLONG`).

A user is allowed to upgrade the label on his or her directory if the directory is empty and the user has write access to the directory.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list.
`PRIV_MAC_DOWNGRADE`	The process is allowed to use this system call to set the security level of the file to a value that is less than or equal to the current security level of the file.
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label.
`PRIV_MAC_UPGRADE`	The process is allowed to use this system call to set the security level of the file to a value that is greater than or equal to the current security level of the file.

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix and is allowed to upgrade or downgrade the security level of the file.

RETURN VALUES

If `setflvl` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setflvl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the path prefix denies search permission.
`EFAULT`	The *fname* argument points outside the process address space.
`EINVAL`	The specified file resides on a nonnative file system.
`ELEVELV`	The requested security level is not authorized for the file system on which the file resides.
`ENAMETOOLONG`	The specified file name is too long.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have permission to upgrade an empty directory.
`ESECADM`	The process does not have appropriate privilege to use this system call.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setflvl` system call

SEE ALSO

`secstat`(2), `setdevs`(2), `setfacl`(2), `setfcmp`(2), `setfflg`(2)

`spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setjob` – Sets job ID

SYNOPSIS

`int setjob (int` *uid*`, int` *sig*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setjob` system call creates a new job by assigning a new job ID and job table entry to the calling process. If successful, the calling process becomes the first process in the job.

The `setjob` system call accepts the following arguments:

uid Specifies the real user ID of the job owner.

sig Specifies the signal to be sent to the job parent when the last process in the job exits. The job parent is defined as the parent of the calling process. If *sig* is 0, no signal is sent on job termination.

A job is a set of one or more processes. Jobs may have resource limits that are enforced by the system. The job ID and all resource limits are inherited by child processes. Only a process with appropriate privilege can use this system call.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_RESOURCE`	The process is allowed to use this system call.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_ID` permbit is allowed to use this system call.

RETURN VALUES

If `setjob` completes successfully, the job ID is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setjob` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EAGAIN`	The job table is full.
`EINVAL`	Invalid *uid* or *sig* argument.

EPERM The process does not have appropriate privilege to use this system call.

SEE ALSO

fork(2), getjtab(2), killm(2), limit(2), nicem(2), signal(2), suspend(2), waitjob(2)

NAME

setlim – Sets user-controllable resource limits

SYNOPSIS

```
#include <sys/category.h>
#include <sys/resource.h>
int setlim (int id, struct resclim *rptr);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setlim system call establishes resource limit values. It accepts the following arguments:

id Specifies the PID, SID, or UID that corresponds to the resclim field resc_category. A 0 indicates the current PID, SID, or UID. Only a process with appropriate privilege can set resource limits of another user, process, or session.

rptr Points to the resclim structure. It includes the following members (for a complete list, see /usr/include/sys/resource.h):

```
struct resclim {
  int  resc_resource;          /* One of:  L_CPU */
  int  resc_category;          /* One of:  C_PROC, C_SESS, C_UID, C_SESSPROCS */
  int  resc_type;              /* One of:  L_T_ABSOLUTE, L_T_HARD, L_T_SOFT */
  int  resc_action;            /* One of:  L_A_TERMINATE, L_A_CHECKPOINT */
  int  resc_used;              /* Current amount of resource used */
  int  resc_value[R_NLIMTYPES];/* Current resource limit value for each of: */
                               /* L_T_ABSOLUTE, L_T_HARD, L_T_SOFT */
};
```

To set a limit value, all resclim fields must be set to either a value or a null. To set a value to be unlimited, use CPUUNLIM. To set a value to be null, use NULL.

The following describes each of the fields in the resclim structure and their acceptable values.

Field	Description
resc_resource	Represents the resource for which a limit is to be established. Currently, only central processing unit (CPU) resources are supported; therefore, the value of resc_resource must be L_CPU.
resc_category	Identifies which category of resource is to be set. The resc_category determines if the *id* argument is a PID, SID, or UID. Acceptable values are: C_PROC, C_SESS, C_UID, and C_SESSPROCS. The resc_category of C_SESSPROCS requires a SID. A short description follows:

Value	Description
C_PROC	Sets process limits
C_SESS	Sets session limits
C_UID	Sets user limits
C_SESSPROCS	Sets default process limits for the session

resc_type
: Identifies the type of limit to be set. Acceptable values are: L_T_ABSOLUTE, L_T_HARD, and L_T_SOFT. Only a process with appropriate privilege can set L_T_ABSOLUTE limits.

resc_action
: Determines, when a hard limit is reached, whether the process is checkpointed before termination. Acceptable values are: NULL, L_A_TERMINATE or L_A_CHECKPOINT. When the resc_action field is set to L_A_TERMINATE or L_A_CHECKPOINT, the resc_type must be L_T_HARD.

resc_used
: Is not used with the setlim system call. The acceptable value is NULL.

resc_value[R_NLIMTYPES]
: Is an array of three words that contain the absolute, hard, and soft limit values. To set the absolute limit, field resc_value[L_T_ABSOLUTE] must be set. Only a process with appropriate privilege can set absolute limits. To set hard limits, the field resc_type must be set to L_T_HARD and a value must be placed in resc_value[L_T_HARD]. To set soft limits, field resc_type must be set to L_T_SOFT and a value must be placed in resc_value[L_T_SOFT]. The values in resc_value[R_NLIMTYPES] for resc_resource L_CPU must be in clocks. Only one of the following can be set with each setlim system call: resc_value[L_T_ABSOLUTE], resc_value[L_T_HARD], or resc_value[L_T_SOFT].

NOTES

The following mandatory access control (MAC) write check is performed based on the resc_category parameter:

Parameter	Description of check
C_PROC	Against the specified process
C_SESS	Against the session leader
C_SESSPROCS	Against each process in the session
C_UID	No MAC check performed

That is, the active security label of the calling process must be equal to the security label of each process where MAC write access is being verified.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_WRITE`	The calling process is granted write permission to every affected process via the security label.
`PRIV_POWNER`	The calling process is allowed to set the resource limits of another user, process, or session.
`PRIV_RESOURCE`	The calling process is allowed to set absolute limits.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to set the resource limits of another user, process, or session. The super user is allowed to set absolute limits. If the `PRIV_SU` configuration option is enabled, the super user is granted write permission to every affected process via the security label.

RETURN VALUES

If `setlim` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setlim` system call fails and no resource limits are set if one of the following conditions occurs:

Error Code	**Description**
`EFAULT`	The address specified for *rptr* is not valid.
`EINVAL`	One of the arguments contains a value that is not valid.
`EPERM`	The calling process does not have appropriate privilege to set absolute limits.
`EPERM`	The calling process does not have appropriate privilege to set resource limits of another user, process, or session.
`EPERM`	An attempt is made to change a limit on a system process; this is not allowed.
`ESRCH`	No processes are found that match the request.

SEE ALSO

`getlim`(2)

`nlimit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`nlimit`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`NLIMIT`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`setpal` – Sets the privilege assignment list (PAL) and privilege sets of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int setpal (char *path, pal_t *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setpal` system call sets the privilege assignment list (PAL) and privilege sets of a file using the information in the buffer. The calling process must have `PRIV_SETFPRIV` in its effective privilege set, and must either own the file or have `PRIV_FOWNER` in its effective privilege set. The calling process must have MAC write access to the file or have `PRIV_MAC_WRITE` in its effective privilege set. The caller can change the state of privileges in the file's allowed, forced, or set-effective privilege sets only when those privileges are also in the caller's permitted privilege set.

If the `PRIV_SU` configuration option is enabled, then any process with effective user ID 0 meets all the requirements specified in the previous paragraph.

The `setpal` system call accepts the following arguments:

path Specifies the file for which the PAL and privilege sets are set.

buf Points to the buffer that contains the PAL and privilege set information.

bufsize Indicates the size of the buffer in bytes.

RETURN VALUES

If `setpal` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setpal` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	A component of the *path* prefix denies search permission.
`EACCES`	The caller is denied MAC write access to the file.
`EFAULT`	The *buf* or *path* argument points outside the address space of the process.
`EINVAL`	The *bufsize* argument specifies an invalid value.

`EINVAL`	The contents of the supplied PAL is invalid.
`ENAMETOOLONG`	The supplied file name is too long.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the *path* prefix is not a directory.
`EPERM`	The process is not the file owner and does not have appropriate privilege.
`EROFS`	The affected file system is a read-only file system.
`ESECADM`	The process does not have appropriate privilege to use this system call.

SEE ALSO

`fgetpal`(2), `fsetpal`(2), `getpal`(2)

NAME

`setpgid` – Sets process-group-ID for job control

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `setpgid` system call is used to join either an existing process group or create a new process group within the session of the calling process. The process-group-ID of a session leader does not change.

The `setpgid` system call accepts the following arguments:

pid Specifies the existing process ID.

pgid Specifies the new process ID.

On successful completion, the process-group-ID of the process with a process ID that matches *pid* is set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process is used; if *pgid* is 0, the process ID of the process indicated by *pid* is used.

RETURN VALUES

If `setpgid` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setpgid` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	The value of *pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the `exec`(2) functions.
`EINVAL`	The value of *pgid* is less than 0 or is not a value supported by the implementation.

`EPERM`	The process indicated by *pid* is a session leader. The value of *pid* is valid but matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process. The value of *pgid* does not match the process ID of the process indicated by *pid* and no process with a process group ID exists that matches the value of *pgid* in the same session as the calling process.
`ESRCH`	The value of *pid* does not match the ID of the calling process or of a child of the calling process.

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `setpgid` system call

SEE ALSO

`exec`(2), `getpgrp`(2), `setsid`(2), `tcgetpgrp`(2), `tcsetpgrp`(2)

NAME

setpgrp – Sets process-group ID

SYNOPSIS

```
#include <unistd.h>
int setpgrp (void);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setpgrp system call sets the process-group ID of the calling process to the process ID of the calling process and returns the new process-group ID.

RETURN VALUES

The setpgrp system call returns the value of the new process-group ID.

FORTRAN EXTENSIONS

The setpgrp system call may be called from Fortran as a function:

```
INTEGER SETPGRP, I
I = SETPGRP ( )
```

EXAMPLES

This example shows how to use the setpgrp system call to establish a new process group. (Some system calls in the example are not supported on Cray MPP systems.) The group includes the calling process as well as any of its descendents (in this case, three child processes). As a result of the setpgrp request, the new process group ID (PGID) is the process ID (PID) of the calling process.

Typically, a user's processes terminate when the user logs off because all of the user's processes are usually included in the process group of the user's shell process. In contrast, if this program is initiated as a background process and the interactive user logs off from UNICOS, the process and its descendents will not terminate but continue to execute.

```
#include <unistd.h>

main()
{
     int res;

     setpgrp();    /* establish new process group here */

     res = fork();
     if (res == 0) {
          execl("child1", "child1", 0);
          perror("execl for child1 failed");
          exit(1);
     }

     res = fork();
     if (res == 0) {
          execl("child2", "child2", 0);
          perror("execl for child2 failed");
          exit(1);
     }

     res = fork();
     if (res == 0) {
          execl("child3", "child3", 0);
          perror("execl for child3 failed");
          exit(1);
     }

     /* parent program performs its work here */

}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setpgrp` system call

SEE ALSO

`exec`(2), `fork`(2), `getpid`(2), `intro`(2), `kill`(2), `signal`(2)

NAME

setportbm, getportbm – Sets or gets the kernel memory port bit map

SYNOPSIS

```
#include <sys/types.h>
#include <sys/sysmacros.h>

int setportbm (unsigned long *bitmap);

int getportbm (unsigned long *bitmap);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setportbm system call copies *bitmap* into the kernel memory port bit map, which reflects the well-known reserved port numbers defined in the /etc/services file.

The getportbm system call gets a copy of the port bit map in the kernel memory.

The setportbm and getportbm system calls accept the following argument:

bitmap Points to the bit map to copy into or from the kernel memory. *bitmap* is an array of unsigned long integers. Its declaration should always be as follows:

```
u_long      bitmap[PORTBITMAX];
```

NOTES

Never use the setportbm and getportbm system calls directly. Only the rsvportbm(8) administrator command should set the kernel memory port bit map, and only the bindresvport(3C) and rresvport(3C) library routines should access the port bit map.

Only a super user or a process with PRIV_ADMIN on a least privilege system can use the setportbm system call.

RETURN VALUES

If setportbm or getportbm completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The setportbm system call fails if one of the following error conditions occurs:

Error Code	**Description**
EFAULT	Cannot copy the bit map into the kernel memory.

`EINVAL`	The pointer to the port bit map (*bitmap*) is `NULL`.
`EPERM`	The user is not super user.

The `getportbm` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	Cannot get the bit map from the kernel memory.
`EINVAL`	The pointer to the port bit map (*bitmap*) is `NULL`.

EXAMPLES

The following example shows how to use the `setportbm` and `getportbm` system calls:

```
#include <sys/types.h>
#include <sys/sysmacros.h>

main()
{
      u_long      bitmap[PORTBITMAX];

      setportbm(&bitmap[0]);
      getportbm(&bitmap[0]);
}
```

FILES

`/etc/services`	Contains a list of port numbers

SEE ALSO

`bindresvport`(3C), `rresvport`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`rsvportbm`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`setppriv` – Sets the privilege state of the calling process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int setppriv (priv_proc_t *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setppriv` system call sets the privilege state of the calling process to the state contained in the buffer. This call returns an error if an attempt is made to modify the state of any privilege that is not permitted for the process. This system call does not set the value of the process privilege text.

The `setppriv` system call accepts the following arguments:

buf Specifies the privilege state to be set to the calling process.

bufsize Specifies the size of the buffer in bytes.

RETURN VALUES

If `setppriv` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

If the return value is –1, the privilege state of the calling process is not affected.

ERRORS

The `setppriv` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EFAULT`	The *buf* argument points outside the address space of the process.
`EPERM`	The caller attempted to modify the state of a privilege that did not exist in its permitted privilege set.

SEE ALSO

`getppriv`(2)

NAME

`setregid`, `setegid`, `setrgid` – Sets real or effective group ID

SYNOPSIS

All Cray Research systems:

```
#include <unistd.h>

int setregid (int rgid, int egid);
```

Cray PVP systems:

```
#include <unistd.h>

int setegid (int egid);

int setrgid (int rgid);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setregid` system call sets the real and effective group IDs of the current process to the argument values *rgid* and *egid*, respectively. It accepts the following arguments:

rgid Specifies the real group ID.

egid Specifies the effective group ID.

If *rgid* is –1, the real group ID is not changed; if *egid* is –1, the effective group ID is not changed. The `setegid` call sets the effective group ID of the current process; `setegid(`*egid*`)` is equivalent to the following:

```
setregid(-1, egid)
```

The `setrgid` call sets the real group ID of the current process; `setrgid(`*rgid*`)` is equivalent to the following:

```
setregid(rgid, -1)
```

Processes with appropriate privilege can set their real and effective group IDs to any value. All other processes can change only their effective group ID to their real group ID or their real group ID to their effective group ID.

NOTES

These calls are provided for compatibility reasons; they aid in the porting of code from other systems. Future releases may not support them.

A process with the effective privilege is granted the following ability:

Privilege	Description
PRIV_SETGID	The process may set its real and effective group IDs to any specified value.

If the PRIV_SU configuration option is enabled, the super user may set its real and effective group IDs to any specified value.

RETURN VALUES

If the setregid, setegid, or setrgid calls complete successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

If the following condition occurs, the setregid, setegid, or setrgid system call fails.

Error Code	Description
EPERM	The process does not have appropriate privilege to set its real and effective group IDs to the specified values.

EXAMPLES

The setegid request is generally used in *setgid programs*. A setgid program is one that has had its setgid permission bit (octal 2000) set by the chmod(1) command.

When a user executes a setgid program belonging to another group, the effective group ID and saved group ID of the process is set to the group ID of the group owning the program. It is the process's effective group ID that is checked when access to a file is attempted.

Therefore, a user executing another user's setgid program would be allowed to open files belonging to the other user's group for which the user possibly would not be given access permission by the normal access permission bits. While a process's effective group ID is changed to that of another user's group, UNICOS thinks the process belongs to that other group.

The following program has had its setgid permission bit (octal 2000) set by the chmod(1) command. This program shows a common usage of the setegid request.

```
#include <unistd.h>

main()
{
     int gid, egid;

     gid = getgid();
     egid = getegid();

     printf("real group ID of process (before setegid()) is %d\n", gid);
     printf("effective group ID of process (before setegid()) is %d\n", egid);

     /* Open any files that have restricted access here.  That is, this
        program (assuming it can be executed by any user) needs to open files
        belonging to the same group as the owner of this program but those
        files have no general access permission for any other user.  Assuming
        this program is a setgid program, these open(2) requests are permitted
        since the effective group ID of this process has been changed to the
        group ID of the owner of the program. */

     setegid(getgid());  /* for security reasons, set effective group ID to
                            value of real group ID */

     printf("real group ID of process (after setegid()) is %d\n", getgid());
     printf("effective group ID of process (after setegid()) is %d\n",
                                                             getegid());
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setregid`, `setegid`, and `setrgid` system calls

SEE ALSO

`getgid`(2), `setgid`(2), `setreuid`(2), `setuid`(2)

`chmod`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`setreuid`, `seteuid`, `setruid` – Sets real or effective user ID

SYNOPSIS

```
#include <unistd.h>
int setreuid (int ruid, int euid);
int seteuid (int euid);
int setruid (int ruid);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `setreuid` system call sets the real and effective user IDs of the current process according to the argument values *ruid* and *euid*, respectively. It accepts the following arguments:

ruid Specifies the real user ID.

euid Specifies the effective user ID.

If *ruid* or *euid* is –1, the real or effective user ID remains unchanged. The `seteuid` call sets the effective user ID of the current process; `seteuid(`*euid*`)` is equivalent to the following:

```
setreuid(-1, euid)
```

The `setruid` call sets the real user ID of the current process; `setruid(`*ruid*`)` is equivalent to the following:

```
setreuid(ruid, -1)
```

Processes with appropriate privilege can set their real and effective user IDs to any value. Any other process is restricted to changing only its effective user ID to either its real user ID or saved user ID.

NOTES

These calls are provided for compatibility reasons; they aid in the porting of code from other systems. Future releases might not support these calls; therefore, use `setuid`(2), which will continue to be supported.

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_SETUID`	The process may set its real and effective user IDs to any specified value.

If the `PRIV_SU` configuration option is enabled, the super user may set its real and effective group IDs to any specified value.

RETURN VALUES

If the `setreuid`, `seteuid`, or `setruid` call completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

If the following error condition occurs, the `setreuid`, `seteuid`, or `setruid` system call fails.

Error Code	Description
`EPERM`	The process does not have appropriate privilege to set it real and effective user IDs to the specified values.

BUGS

If NFS block io daemons are running (`biod` for asynchronous write operations) and the write request is handled by a `biod`, the `write`() will appear to succeed. The `biod` will get an error back, but will be unable to return the error to the user, because it was an asynchronous operation. The server is left with an empty file, and the error is listed in the error return following the `close`().

EXAMPLES

The `seteuid` request is generally used in *setuid programs*. A setuid program is one that has had its setuid permission bit (octal 4000) set by the `chmod`(1) command.

When a user executes a setuid program belonging to another user, the effective ID and saved ID of the process is set to the ID of the user owning the program. It is the process's effective ID that is checked when access to a file is attempted.

Therefore, a user executing another user's setuid program would be allowed to open files belonging to the other user for which the user possibly would not be given access permission by the normal access permission bits. While a process's effective ID is changed to that of another user, UNICOS thinks the process belongs to that other user.

The following program has had its setuid permission bit (octal 4000) set by the `chmod`(1) command. This program shows common usages of the `seteuid` request.

```
#include <unistd.h>

main()
{
     int uid, euid;

     uid = getuid();
     euid = geteuid();

     printf("real ID of process (before setuid()) is %d\n", uid);
     printf("effective ID of process (before setuid()) is %d\n", euid);

     /* Open any files that have restricted access here.  That is, this
        program (assuming it can be executed by any user) needs to open files
        belonging to the same user as the owner of this program but those
        files have no general access permission for any other user.  Assuming
        this program is a setuid program, these open(2) requests are permitted
        since the effective Id of this process has been changed to that of the
        owner of the program. */

     seteuid(getuid());  /* for security reasons, set effective ID to value
                            of real ID */

     printf("real ID of process (after setuid()) is %d\n", getuid());
     printf("effective ID of process (after setuid()) is %d\n", geteuid());

     seteuid(euid);      /* set effective ID back to the effective ID the
                            process originally had since another restricted
                            file needs to be opened now */

     /* open the restricted file here */

     seteuid(getuid());   /* for security reasons, set effective ID to
                             value of real ID - will automatically occur
                             when process dies */
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setreuid`, `seteuid`, and `setruid` system calls

SEE ALSO

`getuid`(2), `setregid`(2), `setuid`(2)

`chmod`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`setsid` – Creates session and sets process group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

If the calling process is not a process group leader, the `setsid` system call creates a new session. The calling process is the session leader of this new session, the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group and the only process in the new session.

RETURN VALUES

If `setsid` completes successfully, it returns the process group ID of the calling process; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setsid` system call fails if the following condition occurs:

Error Code	Description
`EPERM`	The calling process is already a process group leader, or the process ID of the calling process equals the process group ID of a different process.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setsid` system call

SEE ALSO

`exec`(2), `exit`(2), `fork`(2), `getpid`(2), `kill`(2), `setpgid`(2), `sigaction`(2)

`tty`(4) *UNICOS File Formats and Special Files Reference Manual,* Cray Research publication SR–2014

NAME

`setsysv` – Sets minimum and maximum level range, authorized compartments, and security auditing options

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysv.h>

int setsysv (struct sysv *buf, int bufsize);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setsysv` system call sets the authorized compartments, and minimum and maximum security level range for the UNICOS system.

The `setsysv` system call accepts the following arguments:

buf Points to a `sysv` structure in which the security values are stored.

bufsize Specifies the size of the `sysv` structure in bytes.

The `sysv` structure includes the following members:

```
short   sy_minlvl;              /* minimum security level */
short   sy_maxlvl;              /* maximum security level */
long    sy_valcmp;              /* authorized compartments */
```

The `setsysv` system call can be used by a properly privileged process to change the selection of the security audit options. To change the options, the options and the `sy_audit_chng` flag are set in the `sysv` structure, which is passed via the *buf* argument.

Only a process with appropriate privilege can use this system call.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_ADMIN`	The process is allowed to use this system call.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call.

The `setsysv` system call sets the security boundary conditions (the minimum and maximum security levels, and authorized compartments) for execution within the system.

The `setsysv` system call does not force termination of tasks initiated at the original system security levels; therefore, the system can still have processes outside of the new level range and authorized compartments.

When the `MLS_OBJ_RANGES` configuration option is enabled, a check is made to ensure that the new minimum and maximum levels and authorized compartments do not conflict with any of the mounted file system labels. Therefore, it is most effective to use `setsysv` at system startup, before the file systems are mounted. The file systems of other companies are treated as if they have a security label of a maximum and minimum security level of 0, and no authorized security compartments.

When the `setsysv` system call is used to change the security auditing options, the new option values are saved into the kernel low memory tables (`lowmem.c`)

All `setsysv` requests are recorded in the security log, indicating success or failure.

RETURN VALUES

If `setsysv` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setsysv` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`ECOMPV`	If the `MLS_OBJ_RANGES` configuration option is enabled, and the requested authorized compartments are not within the authorized UNICOS system set.
`ECOMPV`	The requested authorized compartments conflict with those of a mounted file system.
`EFAULT`	The *buf* argument points outside the process address space.
`EINVAL`	The *bufsize* argument is less than the size of the `sysv` structure. If *bufsize* is greater than the size of the `sysv` structure, *bufsize* is bounded silently by the actual size.
`EINVAL`	The requested minimum security level is greater than the requested maximum security level.
`ESECADM`	The process does not have appropriate privilege to use this system call.
`ESYSLV`	The requested minimum and maximum security level range falls outside the allowable UNICOS system range.
`ESYSLV`	If the `MLS_OBJ_RANGES` configuration option is enabled, and the requested minimum and maximum security level range conflicts with that of a mounted file system.

FILES

`/usr/include/sys/param.h`	Defines configuration files
`/usr/include/sys/sysv.h`	Defines structure for system security values
`/usr/include/sys/types.h`	Contains types required by ANSI X3J11

SEE ALSO

`getsysv`(2)

`spset`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setucat` – Sets active categories of a process

SYNOPSIS

```
#include <unistd.h>
int setucat (long cat);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setucat` system call sets the active categories of the process to the value specified by the category bit mask. The category bit mask is the union of bit values corresponding to each category to be activated. The requested categories must be authorized for the process. A process with appropriate privilege can set its active categories to any value within the authorized category range of the system.

The `setucat` system call accepts the following argument:

cat Specifies the value of the category bit mask, which is used to set the active categories of the process.

NOTES

All `setucat` requests are recorded in the security log, indicating success or failure.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_RELABEL_SUBJECT`	The process is allowed to set its active categories to any value within the authorized category range of the system.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to set its active categories to any value within the authorized category range of the system.

RETURN VALUES

If `setucat` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setucat` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EINTCATV`	The requested categories are not authorized for use on the UNICOS system.

`EINTCATV`	The requested categories are not a subset of the caller's authorized categories, and the process does not have appropriate privilege.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setucat` system call

SEE ALSO

`getusrv`(2), `setucmp`(2), `setulvl`(2), `setusrv`(2)

`setucat`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setucmp` – Sets active compartments of the process

SYNOPSIS

```
#include <unistd.h>
int setucmp (long cmp);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setucmp` system call sets the active compartments of the process to the value specified by the compartment bit mask. The compartment bit mask is the union of bit values corresponding to each compartment to be activated. The *cmp* argument must include all compartments that were active for the process prior to this call.

Each compartment specified by *cmp* must be authorized for the process. A process with appropriate privilege can set its active compartments to any value within the authorized compartment range of the system.

The `setucmp` system call accepts the following argument:

cmp Specifies the value of the compartment bit mask, which is used to set the active compartments of the process.

NOTES

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_RELABEL_SUBJECT`	The process is allowed to set its active compartments to any value within the authorized compartment range of the system.
`PRIV_MAC_RELABEL_SUBJECT`	The process is not restricted to the login shell process.
`PRIV_MAC_RELABEL_SUBJECT`	The process environment may contain additional background processes.
`PRIV_MAC_RELABEL_SUBJECT`	The process is allowed to override security compartment access violations with open files.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to set its active compartments to any value within the authorized compartment range of the system. The super user is not restricted to the login shell process. The super user environment may contain additional background processes. The super user is allowed to override security compartment access violations with open files.

Because of standard I/O buffering, data may be lost when a subject's security label is changed. This occurs if the subject does not have MAC access to the file when the buffer is flushed.

RETURN VALUES

If `setucmp` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setucmp` system call fails if one of the following error conditions occurs:

Error Code	Description
`EMANDV`	The requested compartments are not authorized for use on the UNICOS system.
`EMANDV`	The requested compartments are not a subset of the caller's authorized compartments, and the process does not have appropriate privilege.
`EMANDV`	Activating the requested compartments creates an access violation with existing open files (open character special files owned by the caller are a special case), and the process does not have appropriate privilege.
`EMANDV`	The request is not issued from the login shell process, and the process does not have appropriate privilege.
`EMANDV`	There was more than one multitask group in the job (there are background processes), and the process does not have appropriate privilege.
`EMANDV`	The requested compartment set does not include all compartments that were active prior to this call, and the process does not have appropriate privilege.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setucmp` system call

SEE ALSO

`getusrv`(2), `setucat`(2), `setulvl`(2), `setusrv`(2)

`setucmp`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setuid`, `setgid` – Sets user or group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int setuid (uid_t uid);

int setgid (gid_t gid);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `setuid` system call sets the real user ID, effective user ID, and saved user ID of the calling process; `setgid` sets the real group ID, effective group ID, and saved group ID of the calling process. The `setuid` and `setgid` system calls accept the following arguments:

uid Specifies the real user ID, effective user ID, and saved user ID.

gid Specifies the real group ID, effective group ID, and saved group ID.

The following conditions determine the setting of an ID. They are checked in the order given, and the first condition that is true is the one that applies:

- If the process has appropriate privilege, the real, effective, and saved IDs are all set to *uid* (or *gid*).
- If *uid* is equal to either the real user ID or the saved user ID, the effective user ID is set to *uid*.
- If *gid* is equal to either the real group ID or the saved group ID, the effective group ID is set to *gid*.

NOTES

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_SETGID`	The process may set the real group ID, effective group ID, and saved group ID.
`PRIV_SETUID`	The process may set the real user ID, effective user ID, and saved user ID.

If the `PRIV_SU` configuration option is enabled, the super user may set the real, effective, and saved IDs.

RETURN VALUES

If `setuid` or `setgid` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setuid` or `setgid` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EINVAL`	The *uid* is out of range.
`EPERM`	The real user or group ID of the calling process is not equal to *uid* or *gid*, and the process does not have appropriate privileges.

FORTRAN EXTENSIONS

The `setuid` system call can be called from Fortran as a function:

```
INTEGER uid, SETUID, I
I = SETUID (uid)
```

The `setgid` system call can be called from Fortran as a function:

```
INTEGER gid, SETGID, I
I = SETGID (gid)
```

BUGS

If a shell script is made set *uid* or set *gid* and starts with "#!" and the name of the shell to execute the shell script, `exec`(2) in the kernel should execute the shell with the specified effective *gid* or effective *gid*. Instead, `exec`(2) checks the shell for set *uid* and set *gid*, even though the set *uid* and set *gid* of the shell script should take precedence.

EXAMPLES

The `setuid` request is generally used in *setuid programs*. A setuid program is one that has had its setuid permission bit (octal 4000) set by the `chmod`(1) command.

When a user executes a setuid program belonging to another user, the effective ID and saved ID of the process is set to the ID of the user owning the program. It is the process's effective ID that is checked when access to a file is attempted.

Therefore, a user executing another user's setuid program would be allowed to open files belonging to the other user for which the user possibly would not be given access permission by the normal access permission bits. While a process's effective ID is changed to that of another user, UNICOS thinks the process belongs to that other user.

The following program has had its setuid permission bit (octal 4000) set by the `chmod`(1) command. This program shows common usages of the `setuid` request. It behaves differently if the owner is a privileged user.

```
#include <unistd.h>

main()
{
     int uid, euid;

     uid = getuid();
     euid = geteuid();

     printf("real ID of process (before setuid()) is %d\n", uid);
     printf("effective ID of process (before setuid()) is %d\n", euid);

     /* Open any files that have restricted access here.  That is, this
        program (assuming it can be executed by any user) needs to open files
        belonging to the same user as the owner of this program but those
        files have no general access permission for any other user.  Assuming
        this program is a setuid program, these open(2) requests are permitted
        since the effective ID of this process has been changed to that of the
        owner of the program. */

     setuid(getuid());  /* for security reasons, set effective ID to value
                           of real ID */

     printf("real ID of process (after setuid()) is %d\n", getuid());
     printf("effective ID of process (after setuid()) is %d\n", geteuid());

     setuid(euid);      /* set effective ID back to the effective ID the
                           process originally had since another restricted
                           file needs to be opened now */

     /* open the restricted file here */

     setuid(getuid());   /* for security reasons, set effective ID to
                            value of real ID - will automatically occur
                            when process dies; call fails if program is
                            owned by a privileged user */
}
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `setuid` system call

SEE ALSO

exec(2), getuid(2), intro(2), setregid(2), setreuid(2)

chmod(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

setulvl – Sets the active security level of the process

SYNOPSIS

```
#include <unistd.h>
int setulvl (int level);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The setulvl system call raises the active security level of the calling process. A process with appropriate privilege can raise or lower its active security level to syslow, syshigh, or to any value within the security level range of the system.

The setulvl system call accepts the following argument:

level Specifies the value of the active security level of the calling process. This argument must fall within the authorized security level range of the process.

NOTES

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_MAC_RELABEL_SUBJECT	The process is allowed to raise or lower its active security level to syshigh, syslow, or to any value within the security level range of the system.
PRIV_MAC_RELABEL_SUBJECT	The process is not restricted to the login shell process.
PRIV_MAC_RELABEL_SUBJECT	The process environment may contain additional background processes.
PRIV_MAC_RELABEL_SUBJECT	The process is allowed to override security level access violations with open files.

If the PRIV_SU configuration option is enabled, the super user is allowed to raise or lower its active security level to syslow, syshigh, or to any value within the security level range of the system. The super user is not restricted to the login shell process. The super user environment may contain additional background processes. The super user is allowed to override security level access violations with open files.

Because of standard I/O buffering, data may be lost when a subject's security label is changed. This occurs if the subject does not have MAC access to the file when the buffer is flushed.

RETURN VALUES

If `setulvl` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setulvl` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EMANDV`	The requested level is not authorized for use on the UNICOS system.
`EMANDV`	The requested level is not within the caller's authorized security level range, and the process does not have appropriate privilege.
`EMANDV`	The requested level is less than the current active security level of the process, and the process does not have appropriate privilege.
`EMANDV`	Changing to the requested level creates an access violation with existing open files (open character special files owned by the caller are a special case), and the process does not have appropriate privilege.
`EMANDV`	The request is not issued from the login shell process, and the process does not have appropriate privilege.
`EMANDV`	There was more than one multitask group in job (there are background processes), and the process does not have appropriate privilege.

FILES

`/usr/include/unistd.h`	Contains C prototype for the `setulvl` system call

SEE ALSO

`getusrv`(2), `setucat`(2), `setucmp`(2), `setusrv`(2)

`setulvl`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`setusrv` – Sets security validation attributes of the process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/usrv.h>

int setusrv (struct usrv *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `setusrv` system call sets security validation attributes for a process.

The `setusrv` system call accepts the following argument:

buf Points to a `usrv` structure in which the attribute values are stored.

A `usrv` structure includes the following members:

```
short   sv_minlvl;      /* minimum security level */
short   sv_maxlvl;      /* maximum security level */
long    sv_valcmp;      /* authorized compartments */
long    sv_savcmp;      /* TFM_EXEC command saved compartments (not used)*/
long    sv_actcmp;      /* active compartments */
short   sv_permit;      /* permissions */
short   sv_actlvl;      /* active security level */
short   sv_savlvl;      /* TFM_EXEC saved security level (not used) */
short   sv_intcls;      /* active integrity class (not used) */
short   sv_maxcls;      /* maximum integrity class (not used) */
long    sv_intcat;      /* active categories */
long    sv_valcat;      /* authorized categories */
struct  {               /* saved integrity parameters over TFM_EXEC
                           (not used) */
        int     actcls  :32;    /* integrity class before TFM_EXEC
                                   (not used) */
        int     actcat  :32;    /* active category before TFM_EXEC
                                   (not used) */
} sv_savint;
int     sv_audit_off    :1;     /* audit on/off flag */
int     sv_audit_chng   :1;     /* audit change flag */
```

A process can use this system call to expand or constrict its active and authorized security attributes. Any process can constrict its authorized security attributes (minimum and maximum security level range, authorized compartments, authorized categories, and so on). Only an appropriately privileged process can expand its authorized security attributes or modify its active security attributes.

A process can enable or disable kernel auditing of its activities by setting the `sv_audit_chg` flag and setting/clearing the `sv_audit_off` flag. Any process can enable kernel auditing for itself. Only an appropriately privileged process can disable kernel auditing of its activities.

NOTES

The login program sets the active security level to the user's default security level with a `setulvl`(2) system call immediately after the `setusrv` call.

All `setusrv` requests are recorded in the security log, indicating success or failure.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to change the state of the `usrtrap` permission.
`PRIV_AUDIT_CONTROL`	The process is allowed to disable kernel auditing of its activities.
`PRIV_MAC_RELABEL_SUBJECT`	The process is allowed to expand its authorized security attributes and to set its active security attributes.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to expand its authorized security attributes and to set its active security attributes. A trusted process is allowed to change the state of the `usrtrap` permission. The super user is allowed to disable kernel auditing of its activities.

Because of standard I/O buffering, data may be lost when a subject's security label is changed. This occurs if the subject does not have MAC access to the file when the buffer is flushed.

RETURN VALUES

If `setusrv` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `setusrv` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`ECOMPV`	The requested active compartments are not authorized for the process.
`EFAULT`	The *buf* argument points outside the process address.
`EINTCATV`	The requested authorized categories include the `archive` category.
`EINTCATV`	The requested active categories are not valid for the process.

`EINTCLSV`	The requested maximum class is not equal to or greater than the authorized minimum class.
`EINTCLSV`	The requested active class is not within the minimum and maximum classes for this process.
`ESYSLV`	The requested minimum level is greater than the requested maximum level.
`ESYSLV`	The requested minimum and maximum level range is not included in the UNICOS system minimum and maximum level range.
`ESYSLV`	The requested active level is not within the minimum and maximum levels for this process.

Additionally, when called by a process without appropriate privilege, `setusrv` fails if one of the following error conditions occurs:

Error Code	**Description**
`ECOMPV`	Attempt was made to expand the authorized compartments.
`ECOMPV`	Attempt was made to change active compartments.
`EINTCATV`	Attempt was made to expand authorized categories.
`EINTCATV`	Attempt was made to change active categories.
`EINVAL`	Attempt was made to expand permissions.
`ESYSLV`	Attempt was made to expand the authorized security level range.
`ESYSLV`	Attempt was made to change active security level.

If the requested minimum and maximum security levels are outside those authorized for the UNICOS system, they are set within the bounds of the system.

If the requested valid compartments, categories, or permissions are outside those authorized for the UNICOS system, they are set within the bounds of the system.

If the calling process does not have `suidgid` permission, the file creation mask of the process is set to disallow creation of setuid or setgid files.

If the calling process has no permissions, or has only user permissions, the process is assigned only the user permissions from the requested set. If the calling process has at least one nonuser permission, `setusrv` sets the process' permissions to the requested value.

When called by a process without appropriate privilege, `setusrv` sets the security labels of open character special files (ttys) to the process' active security label.

SEE ALSO

`setulvl`(2), `getusrv`(2)

`setusrv`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

shmat – Attaches shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
void *shmat (int shmid, void *shmaddr, int shmflg);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an ENOSYS error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The shmat system call attaches the shared memory segment associated with the shared memory identifier. It accepts the following arguments:

shmid Specifies a shared memory segment.

shmaddr Specifies the address of the shared memory segment.

shmflg Specifies a flag value.

The segment is attached to the address specified by one of the following criteria:

- If *shmaddr* is a null pointer, the segment is attached at the first available address as selected by the system.
- If *shmaddr* is not a null pointer and *shmflg*&SHM_RND is not 0, the segment is attached at the address given by *shmaddr* – (*shmaddr* modulus SHMLBA).
- If *shmaddr* is not a null pointer and *shmflg*&SHM_RND is 0, the segment is attached at the address given by *shmaddr*. *shmaddr* must be aligned (on a MEMKLIK boundary).

The segment is attached for reading if *shmflg*&SHM_RDONLY is not 0 and the calling process has read permission. Otherwise, if *shmflg*&SHM_RDONLY is 0 and the process has read and write permission, the segment is attached for reading and writing.

NOTES

If the user has persistence permission, shared memory segments will remain in the system. If the user does not have persistence permission, and does not explicitly remove segments created, these segments are removed from the system when the session terminates or after the final detach, if attached by processes from another session.

The user must explicitly remove shared memory segments after the last reference to them has been removed.

The alignment requirement, which varies in different machines, is determined by the mapping size of the memory system. (To remain XPG4 compliant, `SHMLBA` is expressed as a byte value on UNICOS systems. This allows it to be used in expressions passed into `shmget`(2) to specify a size.)

Processes that have attached shared memory segments cannot be checkpointed or restarted; a checkpoint operation fails with error `ESHMA`.

A process is granted read permission to a shared memory segment only if the active security label of the process is greater than or equal to the security label of the shared memory segment, and the process is granted read access by the shared memory segment access control list (ACL) (if one is assigned).

A process is granted write permission to a shared memory segment only if the active security label of the process is equal to the security label of the shared memory segment, and the process is granted write access by the shared memory segment ACL (if one is assigned).

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_MAC_READ`	The process is considered to meet the security label requirements for being granted read permission to a shared memory segment.
`PRIV_MAC_WRITE`	The process is considered to meet the security label requirements for being granted write permission to a shared memory segment.
`PRIV_DAC_OVERRIDE`	The process is considered to meet the permission mode and ACL requirements for being granted read and write permission to a shared memory segment.

If the `PRIV_SU` configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above. The super user is granted read and write permission to a shared memory segment.

RETURN VALUES

If `shmdt` completes successfully, the value of the `shm_nattch` field in the data structure associated with the shared memory ID of the attached shared memory segment is incremented and a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `shmat` system call fails and does not attach the shared memory segment if one of the following error conditions occurs:

Error Code	Description
`EACCES`	Operation permission is denied to the calling process (see `ipc`(7)).
`EINVAL`	The *shmid* argument is not a valid shared memory identifier.
`EINVAL`	The *shmaddr* argument is not equal to 0, and the value of *shmaddr* – (*shmaddr* modulus `SHMLBA`) is an illegal address for attaching shared memory.
`EINVAL`	The *shmaddr* argument is not equal to 0, *shmflg*&`SHM_RND` is equal to 0, and the value of *shmaddr* is an illegal address for attaching shared memory.
`EMFILE`	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
`ENOMEM`	The available data space is not large enough to accommodate the shared memory segment.
`ENOSYS`	Shared memory operations are permitted only on the CRAY T90 series.

FILES

`/usr/include/sys/shm.h` Contains shared memory data structures and macros

SEE ALSO

`exec`(2), `exit`(2), `fork`(2), `shmctl`(2), `shmdt`(2), `shmget`(2)

`ipc`(5), `shm`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`shmctl` – Provides shared memory control operations

SYNOPSIS

```
#include <sys/shm.h>
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an `ENOSYS` error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The `shmctl` system call provides a variety of shared memory control operations. It accepts the following arguments:

shmid Specifies the shared memory identifier.

cmd Specifies a shared memory control operation. The following are valid *cmd* values.

`IPC_STAT` Places the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in the include file `sys/shm.h` (see `shm`(5)). This command requires read permission.

`IPC_SET` Sets the value of members of the `shmid_ds` data structure associated with *shmid*. It sets the value of the following members to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode     /* low-order 9 bits */
```

The `IPC_SET` command can be executed only by a process that has an effective user ID equal to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*.

`IPC_RMID` Removes the shared memory identifier specified by *shmid* from the system and destroys the shared memory segment and `shmid_ds` data structure associated with *shmid*. The `IPC_RMID` command can be executed only by a process that has an effective user ID equal to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*.

`IPC_SETACL` Sets the access control list (ACL) on the shared memory identifiers specified by *shmid*. The `ipc_perm` structure within the `shmid_ds` structure pointed to by *buf* contains a pointer, `ipc_acl`, to an `acl_rec` structure with the required ACL entries, and a count of those entries, `ipc_aclcount`. If an ACL exists for the shared memory identifier, it is replaced by the one provided with this call. If `ipc_aclcount` is 0, any existing ACL is removed. The calling process must be the owner of the shared memory identifiers specified by *shmid*.

`IPC_GETACL` Retrieves the access control list (ACL) for the shared memory identifier specified by *shmid*. The `ipc_perm` structure within the `shmid_ds` structure pointed to by *buf* contains a pointer, `ipc_acl`, to an `acl_rec` structure where the ACL entries are to be returned. The count of entries to be returned is specified in the `ipc_aclcount` field. If there are more than `ipc_aclcount` entries, only the first `ipc_aclcount` entries are returned. If there are less than `ipc_aclcount` entries, all entries are returned. The return value indicates the number of entries returned. If there is no ACL, the return value is 0. The calling process must have read permission to the shared memory identifiers specified by *shmid*.

`IPC_SETLABEL` Sets the security label on the shared memory identifier specified by *shmid*. The `ipc_perm` structure within the `shmid_ds` structure pointed to by *buf* contains a security level, `ipc_slevel`, and a compartment set, `ipc_scomps`, to be set in the security label on the shared memory identifier. If the shared memory segment is currently attached by any processes, the security label is not altered; a value of −1 is returned and `errno` is set to `EAGAIN`. Only a process with the appropriate privilege can perform this operation.

`SHM_DCACHE` Disables scalar caching of this segment for this process.

`SHM_ECACHE` Enables scalar caching of this segment for this process.

`SHM_ICACHE` Invalidates the scalar cache of each CPU currently running a process with the specified segment attached and cached.

`SHM_LOCK` Locks the shared memory segment specified by *shmid* in memory. This command can be executed only by a process with the appropriate privilege.

`SHM_UNLOCK` Unlocks the shared memory segment specified by *shmid*. This command can be executed only by a process with the appropriate privilege.

buf Points to a structure.

NOTES

If the user has persistence permission, shared memory segments will remain in the system. If the user does not have persistence permission, and does not explicitly remove segments created, these segments are removed from the system when the session terminates or after the final detach, if attached by processes from another session.

The user must explicitly remove shared memory segments after the last reference to them has been removed.

If the kernel list of processes caching each segment becomes corrupted, all processes with that segment attached will be sent the `SIGSMCE` signal. The default action is termination.

A process is granted read permission to a shared memory identifier only if the active security label of the process is greater than or equal to the security label of the shared memory identifier, and the process is granted read access by the shared memory identifier ACL (if one is assigned). This applies to the `IPC_STAT` and `IPC_GETACL` operations.

The `IPC_SET`, `IPC_RMID`, and `IPC_SETACL` operations require that the active security label of the process is equal to the security label of the shared memory identifier.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_MAC_READ`	The process is considered to meet the security label requirements for being granted read permission to a shared memory identifier.
`PRIV_MAC_WRITE`	The process is considered to meet the security label requirements for performing an `IPC_SET`, `IPC_RMID`, or `IPC_SETACL` operation.
`PRIV_DAC_OVERRIDE`	The process is considered to meet the permission mode and ACL requirements for being granted read permission to a shared memory identifier.
`PRIV_FOWNER`	The process is considered to meet the shared memory identifier ownership requirements for the `IPC_SET`, `IPC_RMID`, and `IPC_SETACL` operations. The process is also permitted to lock and unlock a shared memory segment.

If the `PRIV_SU` configuration option is enabled, the super user is granted the same abilities as all effective privileges shown above.

The super user is considered the owner of a shared memory identifier, and is granted read permission to that shared memory identifier. The super-user is also permitted to lock and unlock a shared memory segment.

RETURN VALUES

If `shmctl` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `shmctl` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	The *cmd* argument is equal to `IPC_STAT` and the calling process does not have read permission (see `shm`(5)).
`EACCES`	The *cmd* argument is `IPC_GETACL` and the calling process does not have read permission.
`EAGAIN`	The *cmd* argument is `IPC_SETLABEL` and the shared memory segment is currently attached by one or more processes.
`EFAULT`	The *buf* argument points to an illegal address.
`EFAULT`	The *cmd* argument is `IPC_SETACL` or `IPC_GETACL`, and the `ipc_acl` field in *buf* points to an illegal address.
`EINVAL`	The *shmid* argument is not a valid shared memory identifier.
`EINVAL`	The *cmd* argument is not a valid command.
`EINVAL`	The *cmd* argument is `IPC_SET`, and `shm_perm.uid` or `shm_perm.gid` is not valid.
`EINVAL`	The *cmd* argument is `IPC_SETACL` and one of the following is true: • The `ipc_aclcount` field in *buf* is 0, but there is no ACL associated with *shmid*. • The `ipc_aclcount` field in *buf* is less than 0 or greater than 256. • The ACL supplied failed validation.
`ENOMEM`	The *cmd* argument is equal to `SHM_LOCK` and there is not enough memory.
`ENOMEM`	The *cmd* argument is `IPC_SETACL` and no memory was available to store the ACL. The command should be retried at a later time.
`ENOSYS`	Shared memory operations are permitted only on the CRAY T90 series.
`EPERM`	The *cmd* argument is equal to `IPC_RMID` or `IPC_SET`, and the effective user ID of the calling process is not equal to the process with the appropriate permissions or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*, and the process does not have the appropriate privilege.
`EPERM`	The *cmd* argument is `IPC_SETLABEL`, and the calling process does not have the appropriate privilege.

`EPERM`	The *cmd* argument is `SHM_LOCK` or `SHM_UNLOCK`, and the calling process does not have the appropriate privilege.
`EPERM`	The *cmd* argument is `IPC_SETACL`, and the calling process does not meet ownership requirements and does not have the appropriate privilege.

FILES

`/usr/include/sys/shm.h`	Contains shared memory data structures and macros

SEE ALSO

`exec`(2), `exit`(2), `fork`(2), `shmat`(2), `shmget`(2), `shmdt`(2)

`ipcs`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ipc`(5), `shm`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`shmdt` – Detaches shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
int shmdt (void *shmaddr);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an `ENOSYS` error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The `shmdt` system call detaches the shared memory segment from the calling process's address space. It accepts the following argument:

shmaddr Specifies the address of the shared memory segment.

NOTES

If the user has persistence permission, shared memory segments will remain in the system. If the user does not have persistence permission, and does not explicitly remove segments created, these segments are removed from the system when the session terminates or after the final detach, if attached by processes from another session.

The alignment requirement, which varies on different machines, is determined by the mapping size of the memory system.

RETURN VALUES

If `shmdt` completes successfully, the value of the `shm_nattch` field in the data structure associated with the shared memory ID of the attached shared memory segment is decremented and a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `shmdt` system call fails and does not detach the shared memory segment if one of the following error conditions occurs:

Error Code	**Description**
`EINVAL`	The *shmaddr* argument is not the data segment start address of a shared memory segment.

`EINVAL`	There are outstanding asynchronous I/O operations.
`ENOSYS`	Shared memory operations are permitted only on the CRAY T90 series.

FILES

`/usr/include/sys/shm.h`	Contains shared memory data structures and macros

SEE ALSO

`exec`(2), `exit`(2), `fork`(2), `shmat`(2), `shmctl`(2), `shmget`(2)

`ipc`(5), `shm`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`shmget` – Accesses shared memory identifier

SYNOPSIS

```
#include <sys/shm.h>
int shmget (key_t key, size_t size, int shmflg);
```

IMPLEMENTATION

All Cray Research systems. The interface is supported on all platforms, but invocation will return an `ENOSYS` error for all systems except the CRAY T90 series.

STANDARDS

XPG4

DESCRIPTION

The `shmget` system call returns the shared memory identifier associated with *key*. It accepts the following arguments:

key Specifies the shared memory segment.

size Specifies the shared memory segment size in bytes.

shmflg Specifies a flag value.

A shared memory identifier, associated data structure, and shared memory segment of at least *size* bytes (see `shm`(5)) are created for *key* if one of the following is true:

- *key* is equal to `IPC_PRIVATE`.
- *key* does not already have a shared memory identifier associated with it, and *shmflg*&`IPC_CREAT` is not 0.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- `shm_perm.cuid`, `shm_perm.uid`, `shm_perm.cgid`, and `shm_perm.gid` are set to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of `shm_perm.mode` are set to the low-order 9 bits of *shmflg*. `shm_segsz` is set to the value of *size*.
- `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are set to 0.
- `shm_ctime` is set to the current time.

NOTES

If the calling process has the `ipc_persist` permission bit, then the shared memory identifier will be created as a persistent ID. Persistent shared memory identifiers will not be removed from the system unless a `shmctl`(2) system call with the command `IPC_RMID` or an `ipcrm`(1) command is performed on the ID.

If the calling process does not have this permission bit, then the shared memory identifier will be linked into a list of nonpersistent IDs belonging to the session of which the process is a member. When the last process of the session terminates, all the shared memory identifiers linked to the session will be removed from the system.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_RESOURCE	The process is considered to have the `ipc_persist` permission bit.

If the `PRIV_SU` configuration option is enabled, the super user is granted the same abilities as all effective privileges shown in the preceding list.

The super user is considered to have the `ipc_persist` permission bit.

RETURN VALUES

If `shmget` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `shmget` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	A shared memory identifier exists for *key* but operation permission as specified by the low-order 9 bits of *shmflg* would not be granted (see `ipc`(7)).
EEXIST	A shared memory identifier exists for *key* but both *shmflg*&`IPC_CREAT` and *shmflg*&`IPC_EXCL` are not 0.
EINVAL	The *size* argument is less than the system-imposed minimum or greater than the system-imposed maximum.
EINVAL	A shared memory identifier exists for *key*, but the size of the segment associated with it is less than *size* and *size* is not equal to 0.
EMEMLIM	The request would exceed the limits for the session associated with the calling process.
ENOENT	A shared memory identifier does not exist for *key* and *shmflg*&`IPC_CREAT` is 0.
ENOMEM	A shared memory identifier and associated shared memory segment are to be created, but the amount of available memory is not sufficient to fill the request.
ENOSPC	A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.

`ENOSYS` Shared memory operations are permitted only on the CRAY T90 series.

FILES

`/usr/include/sys/shm.h` Contains shared memory data structures and macros

SEE ALSO

`shmat`(2), `shmctl`(2), `shmdt`(2)

`ipcrm`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`stdipc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`ipc`(5), `shm`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ipc`(7) Online only

NAME

`shutdown` – Shuts down part of a full-duplex connection

SYNOPSIS

`int shutdown (int` *s*`, int` *how*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `shutdown` system call shuts down all or part of a full-duplex connection on the specified socket. It accepts the following arguments:

s Specifies the descriptor for the socket.

how Specifies whether further sends and receives are allowed. The following are valid *how* values:

`0` Further receives are disallowed.

`1` Further sends are disallowed.

`2` Further sends and receives are disallowed.

Unlike the `close`(2) system call, `shutdown` can shut down a socket one direction at a time (send or receive). The `close`(2) system call frees up kernel resources and the socket descriptor, but `shutdown` does not.

NOTES

If some protocols (such as `tcp`(4P)) do a `shutdown` before a `close`(2), the normal termination of a connection is modified.

If the `SOCKET_MAC` option is enabled, the active security label of the process must equal the security label of the socket. Note that `SOCKET_MAC` is part of TCP/IP configurable feature variables list in `uts/cf/Nmakefile`.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_WRITE`	The process is allowed to override the security label restrictions when the `SOCKET_MAC` option is enabled.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security level and compartment restrictions when the `SOCKET_MAC` option is enabled.

RETURN VALUES

If `shutdown` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `shutdown` system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	If the SOCKET_MAC option is enabled, the process does not meet the security label requirements and does not have appropriate privilege.
EBADF	The *s* descriptor is not valid.
EINVAL	An invalid value was specified for *how*.
ENOTSOCK	The *s* descriptor is not a socket.

SEE ALSO

`close`(2), `connect`(2), `socket`(2)

`tcp`(4P) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`sigaction`, `sigvec` – Examines or changes action associated with a signal

SYNOPSIS

```
#include <signal.h>

int sigaction (int sig, const struct sigaction *act,
struct sigaction *oact);

int sigvec (int sig, struct sigvec *vec, struct sigvec *ovec);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `sigaction`)

DESCRIPTION

The `sigaction` system call allows the calling process to examine or specify (or both) the action to be associated with a specific signal.

The `sigaction` system call accepts the following arguments:

sig Specifies the signal. See `signal`(2) for *sig* values.

act or *vec* Specifies the action to be taken when the signal is delivered.

oact or *ovec* Returns the previous signal action.

On Cray MPP systems, the `sigaction` system call examines or changes the signal action only for the PE on which it is called. It has no effect on any other PE of the application.

The `sigaction` structure, which describes an action to be taken, is defined in the `signal.h` header file, and contains the following members:

```
struct sigaction  {
    void (*sa_handler) ();   /* SIG_DFL, SIG_IGN, or pointer to a function */
    sigset_t sa_mask;        /* added to signal mask when in handler */
    int sa_flags;            /* flags to affect behavior of signal */
}
```

If the argument *act* is not null, it points to a structure specifying the action to be associated with the specified signal. If the argument *oact* is not null, the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If *act* is null, signal handling is unchanged by this call; thus the call can be used to inquire about the current handling of a given signal.

The `sa_flags` field specifies a set of flags used to modify the behavior of the specified signal. It is formed by OR'ing together any of the following values (defined in `signal.h`):

`SA_NOCLDSTOP` If set and if *sig* equals `SIGCHLD`, *sig* is not sent to the calling process when its children change state due to job control.

`SA_RESETHAND` If set, the action associated with *sig* is reset to `SIG_DFL` on entry to the signal handler (except for the `SIGILL`, `SIGTRAP`, and `SIGPWR` signals).

`SA_CLEARMASK` If set, *sig* is cleared from the calling process' signal mask on registration.

`SA_CLEARPEND` If set, *sig* is cleared from the set of pending signals on registration.

`SA_NODEFER` If set, *sig* is not added to the calling process' signal mask when entering the signal handler.

`SA_NOCLDWAIT` If set, children of the calling process do not create zombie processes when they terminate.

`SA_WAKEUP` If set, the process is just awakened when *sig* is received and does not enter a signal handler.

`SA_REGMTASK` If set, signal registration is performed for all the tasks in a multitasking group; starting in UNICOS 8.0 this is the default behavior. To get the previous behavior, see the `SA_REGLWP` flag.

`SA_REGLWP` If set, signal registration is performed for the current process; this was the default behavior before UNICOS 8.0. However, it is not recommended that applications depend on this behavior since it may not be supported in later releases.

When a signal is caught by a signal-catching function installed by `sigaction`, a new signal mask is calculated and installed for the duration of the signal-catching function (or until the signal mask is changed explicitly by another system call). This mask is formed by taking the union of the current signal mask and the value of `sa_mask` for the signal being delivered, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested or until one of the `exec`(2) functions is called.

If `sigaction` fails, a new signal handler is not installed.

The `sigvec` system call is provided for 4.3 BSD compatibility. Since the semantics of `sigvec` are equivalent to those of `sigaction` (and the `sigvec` structure has similar members to the `sigaction` structure), this system call is implemented by calling `sigaction` with the same arguments as `sigvec`.

The `sigvec` structure has the following members:

```
struct sigvec {
    void (*sv_handler) ();   /* signal handler */
    int sv_mask;             /* added to signal mask when in handler */
    int sv_flags;            /* use SA_* flags in sigaction(2) */
}
```

RETURN VALUES

If `sigaction` or `sigvec` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `sigaction` or `sigvec` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	A `sigaction` (*act* or *oact*) or `sigvec` (*vec* or *ovec*) argument points to an invalid address.
`EINVAL`	The *sig* argument is an illegal signal number, `SIGKILL`, or `SIGSTOP`.

EXAMPLES

This example shows how to use the `sigaction` system call to prepare for the receipt of a signal. In the following program, the `sigaction` request is anticipating receipt of `SIGINT`.

```
#include <signal.h>

main()
{
    void catch(int signo);
    struct sigaction act, oact;

    act.sa_handler = catch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, &oact);

    printf("\nPrevious disposition for signal SIGINT (#%d) = %o",
                                                 SIGINT, oact.sa_handler);
    if (oact.sa_handler == SIG_DFL)
          printf(" (Default)\n");
    else if (oact.sa_handler == SIG_IGN)
          printf(" (Ignored)\n");
    else
          printf("\n");

    /* The process performs its work here fully prepared if a SIGINT
       signal should be delivered to this process - if SIGINT signal
       is sent, process is interrupted and control passes to routine "catch". */
}

void catch(int signo)
{
     /* Code to process a SIGINT signal resides here - function returns to
        the point of interruption when complete. */
}
```

SEE ALSO

`exec`(2), `signal`(2), `sigpending`(2), `sigprocmask`(2), `sigsuspend`(2)

`sigsetops`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`sigctl` – Provides generalized signal control

SYNOPSIS

```
#include <signal.h>
int sigctl (int action, int sig, void (*func) (int));
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `sigctl` system call, like `signal`(2), allows the calling process to specify what to do upon receipt of a signal.

The `sigctl` system call accepts the following arguments:

action Specifies the action to be taken when the signal is received.

The simplest, and most common, use of `sigctl` is to set *sig* to the desired signal number and set *action* to one of the three bits:

`SCTL_DEF` Takes a system-defined default action.

`SCTL_IGN` Ignores the signal.

`SCTL_REG` Registers to catch the signal.

In this case, *func* contains the address of the signal-catching function or 0. If *func* is set to 0, the process is awakened when the signal occurs, but no signal-catching function is called.

Previously, the following actions provided additional control over the action taken.

`SCTL_KIL`
`SCTL_DMP`
`SCTL_STOP`
`SCTL_CONT`

This control is no longer supported; see the NOTES section for more information. The use of these actions is equivalent to specifying `SCTL_DEF`.

sig Specifies a signal. See `signal`(2) for *sig* values.

func Specifies the address of the signal handler if the action is `SCTL_REG`.

The `sigctl` system call provides additional functionality and control beyond that offered by `signal`(2). The two primary differences with signal-catching in `sigctl` are the following:

- Normally, *func* does not revert to `SIG_DFL`; therefore, the process does not need to re-register the signal-catching function.

- Further signal catching is postponed when the signal-catching function is entered (see `sigon`(3C)).

NOTES

With the introduction of the `sigaction`(2) system call in UNICOS 6.0, the `sigctl` system call has become obsolete. While the `sigaction`(2) interface does not provide a superset of the functionality of `sigctl`, the additional functionality that `sigctl` provides is no longer considered necessary. Because of this change, both the UNICOS MAX and UNICOS versions of `sigctl` are written in terms of `sigaction`(2).

The specific additional functionality provided by `sigctl` is the ability to choose an arbitrary action for any signal. For example, set `SIGINT` to terminate with a core dump or `SIGUSR1` to stop the process. In contrast, `sigaction`(2) only allows the user to ignore, catch with a signal handler, or choose a system-defined "default" action for each signal. For example, `SIGINT` always terminates the process by default, and `SIGABRT` always terminates the process and causes a core dump by default.

When written in terms of `sigaction`(2), calls to `sigctl` with `SCTL_KIL`, `SCTL_DMP`, `SCTL_STOP`, or `SCTL_CONT` as the action are mapped to a `sigaction`(2) call with the handler set to `SIG_DFL`. The other characteristics of `sigctl` are handled as before.

Some additional complexity is involved in returning from the signal-catching function to the point at which the process was interrupted. The C library manages this complexity so that users do not need to understand it. To return to the point of interruption, the operating system must be called to restore the last few registers. Nesting is not limited. To return to the previous environment, a special *action* bit, `SCTL_RET`, is used.

To provide the functionality of `signal` efficiently, where signal-catching usually reverts to killing the process or killing with core dump, there is one additional complexity: When registering a signal-catching function, the process may specify a second bit besides `SCTL_REG`. Before entering the signal-catching function, the status for that signal will be set to one of the following signals as requested, and further signal catching is not postponed.

```
SCTL_IGN
SCTL_KIL
SCTL_STOP
SCTL_CONT
SCTL_DMP
```

RETURN VALUES

If `sigctl` completes successfully, it returns the previous *action* for the specified signal *sig*; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `sigctl` system call fails if the following error condition occurs:

Error Code	Description
`EINVAL`	The *sig* argument is an invalid signal number, including `SIGKILL` or `SIGSTOP`. Also, only `SIGCONT` may be set with the *action* `SCTL_CONT`; `SIGCONT` cannot be registered with `SCTL_KIL`, `SCTL_DMP`, or `SCTL_STOP`.

FORTRAN EXTENSIONS

The `sigctl` system call may be called from Fortran through `fsigctl`(3F).

EXAMPLES

The following example shows how to use the `sigctl` system call to prepare for the receipt of a signal. In this program, the `sigctl` request is anticipating receipt of `SIGINT`.

```
#include <signal.h>

main()
{
     void catch(int signo);

     sigctl(SCTL_REG, SIGINT, catch);

     /* The process performs its work here fully prepared
        if a SIGINT signal should be delivered to this process -
        if SIGINT signal is sent, process is interrupted
        and control passes to routine "catch". */
}

void catch(int signo)
{
     /* Code to process a SIGINT signal resides here -
        function returns to the point of interruption
        when complete. */
}
```

SEE ALSO

`kill`(2), `pause`(2), `ptrace`(2), `sigaction`(2), `signal`(2), `wait`(2)

`kill`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`setjmp`(3C), `sigon`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`fsigctl`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`signal`, `bsdsignal`, `sigset`, `sigignore` – Changes action associated with a signal

SYNOPSIS

```
#include <signal.h>
void (*signal (int sig, void (*func) (int))) (int);
void (*bsdsignal (int sig, void (*func) (int))) (int);
void (*sigset (int sig, void (*func) (int))) (int);
int sigignore (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `signal`)

DESCRIPTION

The `signal`, `bsdsignal`, `sigset`, and `sigignore` system calls allow the calling process to choose the action to be associated with the receipt of a specific signal. All of these calls are implemented in terms of the `sigaction`(2) system call. The *sig* argument specifies the signal, and the *func* argument specifies the choice (`sigignore` has no *func* argument; it implicitly ignores the specified signal).

Valid arguments for the `signal`, `bsdsignal`, `sigset`, and `sigignore` system calls are as follows:

sig Specifies the signal. It can be assigned any one of the signals available on the operating system except `SIGKILL` or `SIGSTOP` (which cannot be caught or ignored): These are listed in the following table:

Signal	Number	Default	Description
`SIGHUP`	1	Exit	Hangup
`SIGINT`	2	Exit	Interrupt
`SIGQUIT`	3	Core	Quit
`SIGILL`	4	Core	Illegal instruction
`SIGTRAP`	5	Core	Trace trap
`SIGABRT`	6	Core	Abort
`SIGERR`	7	Core	Error exit
`SIGFPE`	8	Core	Floating-point exception
`SIGKILL`	9	Exit	Kill (cannot be caught or ignored)
`SIGPRE`	10	Core	Program range error
`SIGORE`	11	Core	Operand range error

Signal	Number	Default	Description
`SIGSYS`	12	Core	Bad argument to system call
`SIGPIPE`	13	Exit	Write on a pipe with no one to read it
`SIGALRM`	14	Exit	Alarm clock
`SIGTERM`	15	Exit	Software termination signal from kill
`SIGIO`	16	Ignore	Input/output possible signal
`SIGURG`	17	Ignore	Urgent condition on I/O channel
`SIGCLD`	18	Ignore	Death of a child process
`SIGPWR`	19	Ignore	Power failure
`SIGBUFIO`	22	Exit	Reserved for CRI-library use on Cray MPP systems
`SIGRECOVERY`	23	Ignore	Recovery signal (advisory)
`SIGUME`	24	Core	Uncorrectable memory error
`SIGDLK`	25	Core	True deadlock detected (Cray PVP systems)
`SIGCPULIM`	26	Exit	CPU time limit exceeded (see `limit`(2))
`SIGSHUTDN`	27	Ignore	System shutdown imminent (advisory)
`SIGSTOP`	28	Stop	Sendable stop signal not from a tty (cannot be caught or ignored)
`SIGTSTP`	29	Stop	Stop signal from a tty
`SIGCONT`	30	Ignore	Continue a stopped process
`SIGTTIN`	31	Stop	To reader’s process group on background tty read
`SIGTTOU`	32	Stop	Like `SIGTTIN` for output, if selected
`SIGWINCH`	33	Ignore	Window size changes
`SIGRPE`	34	Exit	Cray PVP register parity error
`SIGWRBKPT`	35	Core	Write breakpoint (CRAY C90 series only)
`SIGNOBDM`	36	Core	Cray PVP binary enabled bidirectional memory (cannot be caught or ignored)
`SIGAMI`	37	Core	CRAY T90 address multiply interrupt
`SIGSMCE`	38	Exit	Shared memory caching error
`SIGINFO`	48	Ignore	Information signal (see `getinfo`(2))
`SIGUSR1`	49	Exit	User-defined signal 1
`SIGUSR2`	50	Exit	User-defined signal 2

The following alternative definitions are also available:

`SIGIOT`	6
`SIGHWE`	6
`SIGEMT`	7
`SIGBUS`	10
`SIGSEGV`	11
`SIGCHLD`	18

Signals 49 through 64 are available for users.

func Specifies the action associated with the signal.

`SIG_DFL` The default actions are outlined in the default column of the signal table. The defaults are as follows:

Exit Upon receipt of the *sig* signal, the receiving process is terminated with all of the consequences outlined in `exit`(2).

Core Upon receipt of the *sig* signal, the receiving process is terminated. A core image is made in the current working directory of the receiving process if the following conditions are met: first, the effective user ID and the real user ID of the receiving process are equal, and second, a file named `core` (or, if extended core file naming is turned on, `core.`*pid*) can be written or created.

Two forms of a core image can be created. The system attempts to create a restart file of the process (see `restart`(1)). If this fails, the system creates a core image that cannot be restarted. Both forms describe the state of the process at the point the signal was received, but the restart file allows the user to continue execution under the control of a debugger.

Stop Upon receipt of the *sig* signal, the receiving process is stopped.

Ignore Upon receipt of the *sig* signal, the receiving process ignores it. This default is identical to the action specified by `SIG_IGN`.

`SIG_IGN` The *sig* signal is ignored.

`SIG_HOLD` (`sigset` only) The specified signal is added to the calling process' signal mask.

function address Upon receipt of the *sig* signal, the receiving process executes the signal-catching function pointed to by *func*. The signal number *sig* is passed as the only argument to the signal-catching function.

Upon return from the signal-catching function, the receiving process resumes execution at the point at which it was interrupted.

When a signal that is to be caught occurs during certain system calls (for example, a `read`(2) or `write`(2) system call on a terminal or pipe), the signal-catching function is executed, and then the interrupted system call returns a −1 to the calling process with `errno` set to `EINTR`.

The `SIGKILL`, `SIGSTOP`, and `SIGNOBDM` signals cannot be caught or ignored; also, these signals cannot be blocked by using `sigset` with the `SIG_HOLD` action.

Whenever a process receives a `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal, regardless of the action associated with it, any pending `SIGCONT` signal is discarded.

Whenever a process receives a SIGCONT signal, regardless of the action associated with it, any pending SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal is discarded. In addition, if the process was stopped, it is continued.

NOTES

The signal system call is compatible with the ANSI C standard, and also follows UNIX System V, Release 3.0 semantics. The bsdsignal system call is compatible with the 4.3 BSD signal system call, and is renamed to avoid conflicts with the ANSI routine. The sigset and sigignore system calls are provided for System V3 compatibility; their use is discouraged as they do not belong to any particular standard.

Differences in the semantics of these system calls are described in the following list:

System Call	**Description**
signal	When a signal is received (and the action is to execute a signal handler), the action for that signal is reset to SIG_DFL before entering the handler (except for the SIGILL, SIGTRAP, and SIGPWR signals). Also, when a process registers for a signal, all pending signals of that type are cleared.
bsdsignal, sigset	When a signal is received (and the action is to execute a signal handler), the received signal is added to the process' signal mask before entering the handler.
signal, sigset, sigignore	Setting the action for SIGCLD to SIG_IGN causes any child processes of the calling process to not create zombie processes when they terminate (see exit(2)). If the parent process then does a wait(2), the wait blocks until all of the child processes terminate before returning a value of –1 with errno set to ECHILD.
sigset, sigignore	When a process registers for a signal, that signal is cleared from the calling process' signal mask.

Under UNICOS, signal is implemented as a system call, but the signal(3C) function is also defined to be a part of the ANSI Standard C library. For this reason, this documentation appears both here and in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080.

RETURN VALUES

If signal, bsdsignal, or sigset completes successfully, the previous signal action (*func*) is returned; otherwise, a value of SIG_ERR is returned (defined in header file signal.h), and errno is set to indicate the error.

If sigignore completes successfully, 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The `signal`, `bsdsignal`, `sigset`, or `sigignore` system call fails if the following error condition occurs:

Error Code	Description
`EINVAL`	The *sig* argument is an illegal signal number, `SIGKILL`, or `SIGSTOP`.

FORTRAN EXTENSIONS

The `signal` system call can be called from Fortran as a function:

```
INTEGER sig, FSIGNAL, I
EXTERNAL FUNC
I = FSIGNAL(sig, FUNC)
```

Alternatively, `signal` can be called from Fortran as a subroutine. In this case, the return value of the system call is unavailable.

```
INTEGER sig
EXTERNAL FUNC
CALL FSIGNAL (sig, FUNC)
```

The Fortran program must not specify both the subroutine call and the function reference to `signal` from the same procedure.

EXAMPLES

The following examples illustrate different uses of the `signal` system call.

Example 1: This `signal` request prepares for the receipt of a `SIGINT` signal. When using `signal` to catch signals, the programmer needs to remember to re-register to catch the signal in the signal-handling function, because the signal's default disposition is reinstated before entrance to the handler.

```
#include <signal.h>

main()
{
     void catch(int signo);

     signal(SIGINT, catch);

     /* The process performs its work here fully prepared
        if a SIGINT signal should be delivered to this process -
        if SIGINT signal is sent, process is interrupted and
        control passes to routine "catch". */
}

void catch(int signo)
{
     signal(SIGINT, catch);

     /* Code to process a SIGINT signal resides here -
        function returns to the point of interruption
        when complete. */
}
```

Example 2: This `signal` request in conjunction with a `wait`(2) system call causes a process to wait (delay) until all of its child processes have completed:

```
#include <signal.h>

main()
{
     int ret_val, ret_stat;

     signal(SIGCLD, SIG_IGN);

     /* Parent process forks child processes here and
        performs other work - it then wants to wait
        for all of its child processes to terminate. */

     ret_val = wait(&ret_stat);

     /* Parent process proceeds after completion
        of all child processes. */
}
```

SEE ALSO

`exit`(2), `getinfo`(2), `limit`(2), `read`(2), `restart`(2), `sigaction`(2), `sigpending`(2), `sigprocmask`(2), `sigsuspend`(2), `wait`(2), `write`(2)

`restart`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`sigsetops`(3C), `signal`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`sigpending` – Stores pending signals

SYNOPSIS

```
#include <signal.h>
int sigpending (sigset_t *set);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `sigpending` system call stores the set of signals that are blocked from delivery and pending for the calling process. It accepts the following argument:

set Points to the space where the set of signals is stored. On Cray MPP systems, the `sigpending` system call stores pending signals only for the PE on which it is called. It has no effect on any other PE of the application.

RETURN VALUES

If `sigpending` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

EXAMPLES

The following example shows how to use the `sigpending` system call in a program to determine whether there are any signals currently pending and blocked for this process. If any signals are pending, the program displays the corresponding signal numbers.

```
sigset_t pset;
long i;
int j;

if (sigpending(&pset) == -1) {
     perror("sigpending failed");
     exit(1);
}
printf("sigpending reveals the following signals are pending => ");
printf("%lo\n", pset);
if (pset != 0) {
     printf("   or signals numbered =>  ");
     for (i = 1L, j = 1; i > 0; i <<= 1, j++) {
          if (pset & i) {
               printf("%d  ", j);
          }
     }
     printf("\n");
}
```

SEE ALSO

`sigaction`(2), `signal`(2), `sigprocmask`(2), `sigsuspend`(2)

`sigsetops`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

sigprocmask, sigblock, sigsetmask, sighold, sigrelse – Examines and changes blocked signals

SYNOPSIS

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
int sigblock (int mask);
int sigsetmask (int mask);
mask = sigmask (sig);
int sighold (int sig);
int sigrelse (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to sigprocmask)

DESCRIPTION

The sigprocmask system call examines or changes (or both) the calling process's signal mask.

how Indicates the manner in which the set is changed. It consists of one of the following values, as defined in the header file signal.h:

- SIG_BLOCK — The resulting set is the union of the current set and the signal set to which *set* points.
- SIG_UNBLOCK — The resulting set is the intersection of the current set and the complement of the signal set to which *set* points.
- SIG_SETMASK — The resulting set is the signal set to which *set* points.

set Points to a set of signals that can be used to change the current signal mask. If the *set* argument is null, it does not point to a set of signals.

oset Points to the space in which the previous mask is stored. If the *oset* argument is null, it does not point to this space. If the value of *set* is null, the value of *how* is not significant and the process signal mask is unchanged by this system call; the call can be used to inquire about currently blocked signals.

mask Specifies a set of signals as a bitmask.

sig Specifies a signal. See `signal`(2) for *sig* values.

If any pending unblocked signals exist after a call to `sigprocmask`, at least one of those signals is delivered before `sigprocmask` returns.

It is not possible to block the `SIGKILL` and `SIGSTOP` signals; this is enforced by the system without causing an error to be indicated.

The `sigblock` and `sigsetmask` system calls are provided for 4.3 BSD compatibility, and call `sigprocmask` to actually change the signal mask. The `sigblock` system call adds the signals specified in *mask* to the calling process's signal mask. The `sigsetmask` system call sets the calling process's signal mask to the value of *mask*. The `sigmask` macro creates a signal mask for these system calls; to mask a signal *sig*, use `sigmask`(*sig*).

The `sighold` and `sigrelse` system calls are provided for UNIX System V, Release 3.0, compatibility; they also call `sigprocmask` to actually change the signal mask. The `sighold` system call adds the signal *sig* to the calling process's signal mask; `sigrelse` removes the signal *sig* from the mask.

On Cray MPP systems, the `sigprocmask` system call examines or changes blocked signals only for the PE on which it is called. It has no effect on any other PE of the application.

RETURN VALUES

If `sigprocmask`, `sighold`, or `sigrelse` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

The `sigblock` and `sigsetmask` system calls return the old value of the signal mask.

ERRORS

The `sigprocmask` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EFAULT`	The *set* or *oset* argument points to an address that is not valid.
`EINVAL`	The value of *how* is not equal to one of the defined values.

The `sighold` and `sigrelse` system calls fail if the following error condition occurs:

Error Code	**Description**
`EINVAL`	The *sig* argument is an illegal signal number, `SIGKILL`, or `SIGSTOP`.

EXAMPLES

The following examples illustrate how to use the `sigprocmask`, `sigsetmask`, `sigblock`, and `sighold` system calls.

Example 1: In this program, the `sigprocmask` request blocks and unblocks signals for a process. In other words, the example shows how bits are added and removed from the process's signal hold mask.

```
#include <signal.h>

#define NULL 0

main()
{
     sigset_t set, oset;

     if (sigprocmask(NULL, NULL, &oset) == -1) {
          perror("sigprocmask failed");
          exit(1);
     }
     printf("\nInitial signal mask = %lo\n", oset);

     sigemptyset(&set);       /* clear the signal set */
     sigaddset(&set, SIGINT);
     sigaddset(&set, SIGFPE);
     sigaddset(&set, SIGUSR1);

     if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
          perror("sigprocmask failed");
          exit(2);
     }

     /* Signals SIGINT, SIGFPE, and SIGUSR1 are now blocked
        as well as any other signals blocked prior to the
        sigprocmask request. */

     /* Later, it is needed to unblock one of those signals, SIGFPE. */

     sigdelset(&set, SIGUSR1); /* Modify mask such that SIGFPE */
     sigdelset(&set, SIGINT);  /* can be unblocked */

     if (sigprocmask(SIG_UNBLOCK, &set, NULL) == -1) {
          perror("sigprocmask failed");
          exit(3);
     }
}
```

Example 2: In this program, the `sigsetmask`, `sigblock`, and `sighold` requests manipulate a process's signal hold mask.

```
#include <signal.h>

main()
{
     int ret, mask;

     /* sigsetmask(2) is used to hold signals SIGINT and SIGQUIT -
        all other signal types are not held. */

     mask = sigmask(SIGINT) | sigmask(SIGQUIT);
     printf("initial mask = %lo\n", sigsetmask(mask));

     /* sigblock(2) is used to add signal types SIGFPE and SIGUSR2
        to signal hold mask. */

     mask = sigmask(SIGFPE) | sigmask(SIGUSR2);
     ret = sigblock(mask);
     printf("after sigsetmask, mask = %lo\n", ret);

     ret = sigsetmask(0L);        /* determine current mask */
     printf("after sigblock, mask = %lo\n", ret);
     (void) sigsetmask(ret);      /* restore mask */

     /* sighold(2) is used to add signal type SIGUSR1
        to signal hold mask. */

     (void) sighold(SIGUSR1);
     printf("after sighold, mask = %lo\n", sigsetmask(0L));
}
```

SEE ALSO

`sigaction`(2), `signal`(2), `sigpending`(2), `sigsuspend`(2)

`sigsetops`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`sigsuspend`, `bsdsigpause`, `sigpause` – Releases blocked signals and waits for interrupt

SYNOPSIS

```
#include <signal.h>
int sigsuspend (const sigset_t *sigmask);
int bsdsigpause (int mask);
int sigpause (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `sigsuspend`)

DESCRIPTION

The `sigsuspend` system call replaces the process' signal mask with the set of signals pointed to by the *sigmask* argument and then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

The `sigsuspend`, `bsdsigpause`, and `sigpause` system calls accept the following arguments:

sigmask Specifies a set of signal as a bitmask.

mask Specifies a set of signal as a bitmask.

sig Specifies a signal. See `signal`(2) for *sig* values.

If the action is to terminate the process, `sigsuspend` does not return. If the action is to execute a signal-catching function, `sigsuspend` returns after the signal-catching function returns, with the signal mask restored to the setting that existed prior to the `sigsuspend` call. It is not possible to block the `SIGKILL` and `SIGSTOP` signals; this is enforced by the system without indicating an error.

The `bsdsigpause` system call is equivalent to the 4.3 BSD `sigpause` system call; it is renamed to avoid conflicts with the UNIX System V, Release 3.0, `sigpause` system call. It has the same behavior as `sigsuspend`.

The `sigpause` system call is provided for UNIX System V, Release 3.0, compatibility. It releases the signal *sig*, and suspends the process until an interrupt occurs.

On Cray MPP systems, the `sigsuspend` system call suspends only on the PE on which it is called. It has no effect on any other PE of the application.

RETURN VALUES

Since sigsuspend, bsdsigpause, and sigpause suspend process execution indefinitely, no successful completion return value exists; instead, a value of –1 is always returned, and errno is set to indicate the error.

ERRORS

The sigsuspend, bsdsigpause, and sigpause system calls fail if the following error condition occurs:

Error Code	Description
EINTR	A signal is caught by the calling process, and control is returned from the signal-catching function.

EXAMPLES

This example shows how to use the sigsuspend system call to wait for a signal to be delivered to a process. In particular, it shows how the sigsuspend request suspends the program until the process receives a specific signal (SIGUSR1).

```
#include <signal.h>

main()
{
     struct sigaction act;
     sigset_t set;

     act.sa_handler = catch;
     sigemptyset(&act.sa_mask);
     act.sa_flags = 0;
     sigaction(SIGUSR1, &act, NULL);

     sigfillset(&set);           /* turn on (1) all bits in set - */
     sigdelset(&set, SIGUSR1);   /* except SIGUSR1 */

     /* Process performs work here, but after finishing work and before
        proceeding, it needs to wait for a SIGUSR1 signal to be sent
        from another process. */

     sigsuspend(&set);           /* wait for SIGUSR1 signal */

     /* Work continues here after waiting and catching SIGUSR1 signal. */
}

void catch(int signo)
{
     /* process SIGUSR1 signal here */
```

```
}
```

SEE ALSO

`pause`(2), `sigaction`(2), `signal`(2), `sigpending`(2), `sigprocmask`(2)

`sigsetops`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

slgentry – Makes security log entry

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int slgentry (int type, word *entry);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The slgentry system call makes an entry in the /dev/slog security log. The caller defines the type of the entry to be made and passes the address of the entry. The type is decoded and the entry is cast as the proper security structure before the entry is written to the security log.

The slgentry system call accepts the following arguments:

type Defines the type of the entry to be made.

entry Specifies the address of the entry.

The slgentry system call accepts only a specific subset of valid record types. See slrec(5) for more information on these types.

Only an appropriately privileged process can use this system call.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_AUDIT_WRITE	The process is allowed to use this system call.

If the PRIV_SU configuration option is enabled, the super user is allowed to use this system call.

RETURN VALUES

If slgentry completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The `slgentry` system call fails if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The process specified an *entry* where the length is not valid (that is, less than 0 or larger than the largest allowed `slgentry` record).
`EINVAL`	The process specified an *entry* in which some portion of the *entry* is outside the user's address space.
`ESECADM`	The process does not have appropriate privilege to use this system call.

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `slgentry` system call

SEE ALSO

`slog`(4), `slrec`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`slogdemon`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`socket` – Creates an endpoint for communication

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int af, int type, int protocol);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `socket` system call creates sockets. A socket is an endpoint for communications on a local or a remote host.

The `socket` system call accepts the following arguments:

af Specifies an address family with which addresses specified in later operations that use the socket should be interpreted.

These families are defined in include file `sys/socket.h`. The Internet address family (`AF_INET`), the UNIX address family (`AF_UNIX`), and the ISO address family (`AF_ISO`) are the only families currently recognized by UNICOS.

type Specifies the type of socket to be created. Sockets are typed according to their communications properties. Currently defined types are as follows:

`SOCK_STREAM` Provides sequenced, reliable, two-way, connection-based byte streams. It also provides an auxiliary out-of-band data transmission mechanism.

`SOCK_DGRAM` Supports datagrams, which are connectionless, unreliable messages of a fixed (typically small) maximum length.

`SOCK_RAW` Provides access to internal network interfaces. These sockets are available only to the super user.

protocol Specifies a protocol to be used with the socket. (See `icmp`(4P) for an example.) Usually, only one protocol exists to support a particular socket type, using a given address family. However, many protocols can exist; in which case, you must specify a particular protocol in this argument. The protocol number to use is particular to the communication domain in which communication occurs; see `protocols`(5).

Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a connected state before any data can be sent from or received on it. A connection to another socket is created with a connect(2) or accept(2) call. After the socket is connected, data can be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) system calls. When a message is complete, a close(2) call can be performed. *Out-of-band data*, which is data not sent in sequence with other data, can also be transmitted (as described in send(2)) and received (as described in recv(2)).

The communications protocols used to implement a socket of type SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be transmitted successfully within a reasonable length of time, the connection is considered broken, and system calls using the connection return a –1 value and place the ETIMEDOUT code in the global variable errno. The protocols optionally keep sockets active by forcing transmissions every minute in the absence of other activity. If no response can be elicited on an otherwise idle connection for a extended period (for example, 5 minutes), an error is then indicated. If a process sends on a broken stream, a SIGPIPE signal is raised; this causes naive processes, which do not handle the signal, to terminate.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents specified in send(2) calls. It is also possible to receive datagrams at such a socket by using recvfrom (see the recv(2) man page).

You can use an ioctl(2) call to specify a process group to receive a SIGURG signal when out-of-band data arrives.

The socket call sets up send and receive socket buffers (sockbufs) using a protocol-specific sockbuf space limit (see netvar(8)). The socket call fails and returns an ELIMIT error if this call would cause the user's per-session sockbuf space limit to be exceeded.

Socket-level options control the operation of sockets. These options are defined in the sys/socket.h include file and in the following list. Use setsockopt(2) and getsockopt(2) (see getsockopt(2)) to set and to get options, respectively.

The ioctl(2) system call performs a variety of functions at different levels (socket, interface, and routing). The following is a list of command names for the ioctl(2) system call and a description of the function performed by each command.

Command	**Description**
Socket level:	
FIOASYNC	Sets and clears asynchronous input/output by using SIGIO.
FIONBIO	Sets and clears nonblocking input/output.
FIONREAD	Gets number of bytes available for reading from socket.
SIOCATMARK	Indicates whether any out-of-band data is waiting in the socket (0 means yes; otherwise, no).
SIOCGPGRP	Gets process group to receive SIGIO and SIGURG for this socket.
SIOCSPGRP	Sets process group to receive SIGIO and SIGURG for this socket.

Interface level:

`SIOCGIFADDR`	Gets network address of interface into the `ifr_addr` member to which the data argument points.
`SIOCGIFCONF`	Returns interface configuration information into the `ifconf` structure to which the `ioctl` data argument points.
`SIOCGIFDSTADDR`	Specifies address of the remote host on a point-to-point link in the `ifr_addr` member of the `ifreq` structure to which the data argument points.
`SIOCGIFFLAGS`	Returns interface flags in the `ifr_flags` member of the `ifreq` structure to which the data argument points.
`SIOCSIFADDR`	Sets network address of interface from the `ifr_addr` member of the `ifreq` structure to which the data argument points. Also initializes a routing table entry for the interface (must be `root`).
`SIOCSIFDSTADDR`	Sets network address of the remote node on a point-to-point link from the `ifr_addr` member of the `ifreq` structure to which the data argument (must be `root`) points.
`SIOCSIFFLAGS`	Sets interface flags from the `ifr_flags` member of the `ifreq` structure to which the data argument (must be `root`) points. Flag values are as follows:

```
IFF_UP          0x1     /* interface is up */
IFF_DEBUG       0x4     /* turn on debugging */
IFF_POINTOPOINT 0x10    /* interface is point-to-point link */
IFF_NOTRAILERS  0x20    /* avoid use of trailers */
IFF_NOARP       0x80    /* no address resolution protocol */
```

Network media sublevel:

`HYSETROUTE`	Sets HYPERchannel routing table.
`HYGETROUTE`	Gets HYPERchannel routing table.
`HYSETTYPE`	Sets interface type.
`HYGETTYPE`	Gets interface type.

NOTES

The socket is assigned the active security label of the process.

RETURN VALUES

If `socket` completes successfully, a descriptor that references the socket is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `socket` system call fails if one of the following error conditions occurs:

Error Code	**Description**
EMFILE	Either the per-process descriptor table is full, or the system file table is full.
ELIMIT	The user's socket buffer space limit is exceeded.
ENOBUFS	Buffer space is not available. The socket cannot be created.
EPERM	Permission denied for operation.
EPROTONOSUPPORT	The specified protocol is not supported.

EXAMPLES

Because the `socket` system call is used in both client and server programs along with other networking calls, the following examples are simple client and server programs that illustrate how to use the `socket` request.

Example 1: The client program creates a TCP/IP socket and then attempts to establish a connection between the newly created socket and the socket within the server program on the designated server host. If a connection is successful, the client process sends a string of data to the server process.

```
/*  Client side of client/server socket example.
    Syntax: client hostname portnumber */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

/*       in in.h is this socket structure
 *
 *        Socket address, internet style.
 *
 *        struct sockaddr_in {
 *           short   sin_family;
 *           u_short sin_port;
 *           struct  in_addr sin_addr;
 *           char    sin_zero[8];
 *       };
 */

#define DATA "Test message from client to server."

main(int argc, char *argv[])
{
```

```
        int s;
        struct sockaddr_in dest;       /* destination socket address */
        struct hostent *hp;            /* host structure pointer */

   /* Convert host name into network address */
        hp = gethostbyname(argv[1]);
        bzero((char*)&dest, sizeof(sockaddr_in));
        dest.sin_family = hp->h_addrtype;  /* addr type (AF_INET) */
        bcopy(hp->h_addr_list[0], &dest.sin_addr, hp->h_length);
        dest.sin_port = atoi(argv[2]);


        /* create port */

        if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
             perror("client, cannot open socket");
             exit(1);
        }
        if (connect (s, (struct sockaddr *) &dest, sizeof(dest)) < 0) {
             close(s);
             perror("client, connect failed");
             exit(1);
        }
        write(s, DATA, sizeof(DATA));

        close(s);
        exit(0);
   }
```

Example 2: (Some system calls in this example are not supported on Cray MPP systems.) The server program creates a TCP/IP socket, waits for a client process from some host to attempt a connection, accepts the connection, and then forks a child process to provide the service to the client.

The original (parent) server loops back to look for additional connection attempts while the temporary (child) server reads a string of data sent by the client process.

```
/*  Server side of client-server socket example.
    Syntax: server portnumber & */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

main(int argc, char *argv[])
{
     int s, ns;
     struct sockaddr_in src;          /* source socket address */
     int len=sizeof(src);
     char buf[256];

     /* create port */
     src.sin_family = AF_INET;
     src.sin_port = atoi(argv[1]);
     src.sin_addr.s_addr = 0;

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("server, unable to open socket");
          exit(1);
     }

     while (bind(s, (struct sockaddr *) &src, sizeof(src)) < 0) {
          printf("Server waiting on bind...\n");
          sleep(1);
     }

     listen(s, 5);

     while (1) {
          ns = accept(s,(struct sockaddr *) &src, &len);
          if (ns < 0) {
               perror("server, accept failed");
               exit(1);
          }
```

```
            if (fork() == 0) {
                 /* in child server */
                 close(s);     /* child will use socket ns, parent uses s */
                 read(ns, &buf, sizeof(buf));
                 printf("Server read: %s\n", buf);
                 close(ns);
                 exit(0);
            }
            close(ns);         /* close socket used by child */
       }
  }
```

FILES

`/usr/include/net/route.h`	Route file that contains the `rtentry` structure
`/usr/include/sys/socket.h`	Defines the address families
`/usr/include/sys/types.h`	Defines types of sockets

SEE ALSO

`accept`(2), `bind`(2), `close`(2), `connect`(2), `getsockname`(2), `getsockopt`(2), `ioctl`(2), `listen`(2), `read`(2), `recv`(2), `select`(2), `send`(2), `shutdown`(2), `socketpair`(2), `write`(2)

`icmp`(4P), `protocols`(5), `services`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`netvar`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

socketpair – Creates a pair of connected sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

`int socketpair (int` *af*`, int` *type*`, int` *protocol*`, int` *sv*`[2]);`

IMPLEMENTATION

All Cray Research systems

Implemented only for the UNIX address domain (AF_UNIX)

DESCRIPTION

The socketpair system call was designed to simulate the UNIX pipe mechanism with the use of sockets. The call is very similar to the socket(2) system call. With standard network operations, sockets are created individually using socket(2). To simulate pipe operations, the calling process must create both endpoints for the communication simultaneously. socketpair creates a pair of sockets in one request to the operating system.

The socketpair system call accepts the same *af*, *type*, and *protocol* arguments as socket(2) does. For descriptions of these arguments, see the socket(2) man page. In addition, socketpair accepts the following argument:

sv Specifies an array of two integers (*sv*[0] and *sv*[1]) that receive the descriptors for the new pair of sockets.

NOTES

The socket is assigned the active security label of the process.

RETURN VALUES

If socketpair completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The socketpair system call fails if one of the following error conditions occurs:

Error Code	**Description**
EAFNOSUPPORT	This machine does not support the specified address family.
EFAULT	The *sv* address does not specify a valid part of the process address space.
EMFILE	Too many descriptors are in use by this process.

`EOPNOTSUPP`	The specified protocol does not support creation of socket pairs.
`EPROTONOSUPPORT`	This machine does not support the specified protocol.

FILES

`usr/include/sys/socket.h`	Header file for sockets
`usr/include/sys/types.h`	Header file for types

SEE ALSO

`pipe`(2), `read`(2), `write`(2)

NAME

`ssbreak` – Changes size of secondary data segment

SYNOPSIS

```
#include <unistd.h>
int ssbreak (long incr);
```

IMPLEMENTATION

All Cray Research systems except CRAY J90 series and CRAY EL series

DESCRIPTION

The `ssbreak` system call increases or decreases the size of the secondary data segment (SDS), which is allocated from an area of the SSD solid-state storage device reserved for this purpose at configuration time (see `ssd`(4)). It accepts the following argument:

incr Specifies a count of 4096-byte blocks allocated and deallocated in the SSD.

The `ssbreak` system call changes secondary storage size, as follows:

- If the `ssbreak` system call has a positive *incr* argument, the amount of secondary storage increases by a multiple of the number of blocks that is defined as the unit allocation size. The unit allocation size is the number of 4096-byte blocks that compose the smallest amount of space that a user process can allocate in the SDS; UNICOS is delivered with a unit allocation size of 128 blocks. The unit allocation size can be found as `SDS_WGHT` in `/usr/include/sys/ssd.h`. If the *incr* argument is less than or equal to the number of blocks in one unit, the `ssbreak` system call allocates one whole unit of secondary storage. If the *incr* is larger than the number of blocks in one unit and less than or equal to twice the number of blocks in one unit, two units of secondary storage are allocated, and so on. Because the unit for secondary storage usually is greater than one block, the amount of storage allocated can be greater than that requested.
- If the `ssbreak` system call has a negative *incr* argument, it deallocates secondary storage only in multiples of one unit. The *incr* argument must be larger than or equal to the number of blocks in one unit for deallocation to result.
- If the `ssbreak` system call has an *incr* argument of 0, the size of secondary storage remains the same.

Secondary storage allocated by the `ssbreak` system call is freed on exit of the program.

CAUTIONS

The `ssbreak` system call works only on a system with an SSD in its configuration, and a secondary data segment (SDS) area must be configured as a slice of the SSD.

Using `ssbreak` directly interferes with the operation of `sdsalloc`(3F), which manages the SDS space within a process; `sdsalloc`(3F) should be used instead of `ssbreak`. CRI strongly discourages direct use of `ssbreak` in user programs.

RETURN VALUES

If `ssbreak` completes successfully, the current amount of secondary data storage in blocks is returned to the caller; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `ssbreak` system call fails if the following error condition occurs:

Error Code	**Description**
`ENOMEM`	The request requires more secondary storage than can be satisfied.

FORTRAN EXTENSIONS

The `ssbreak` system call can be called from Fortran as a function:

```
INTEGER incr, SSBREAK, I
I = SSBREAK (incr)
```

The `ssbreak` system call should not be used in a Fortran program that accesses SDS through the `assign`(1) command or auxiliary arrays because the libraries use `sdsalloc`(3F) to control SDS allocation. Using `ssbreak` from Fortran directly conflicts with the SDS management that `sdsalloc`(3F) provides.

EXAMPLES

The following example shows how to use the `ssbreak` system call to request a SDS allocation. In this case, the programmer asks for an area of 10 blocks for the process. Because SDS segment space is allocated in units called the unit allocation size, which is typically configured at 128 blocks, this `ssbreak` request actually allocates the process 128 blocks of SDS space.

```
int size;

if ((size = ssbreak(10L)) == -1) {
     perror("sds allocation error");
     exit(1);
}
else {
     printf("The size of the sds is now %d - 4096-byte blocks\n", size);
}

/* To make use of the allocated SDS area, the program next issues
   ssread and sswrite requests. */

/* When usage of the allocated SDS area is complete, the program
   releases its SDS allocation using ssbreak with negative argument -
   process termination also releases SDS space. */

if ((size = ssbreak(-size)) == -1) {
     perror("sds deallocation error");
     exit(1);
}
else {
     printf("The size of the sds is now %d - 4096-byte blocks\n", size);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `ssbreak` system call

SEE ALSO

`assign`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`sdsalloc`(3F) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`ssd`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`ssread`, `sswrite` – Reads or writes to secondary data segment

SYNOPSIS

```
#include <unistd.h>
int ssread (long pds, long sds, long count);
int sswrite (long pds, long sds, long count);
```

IMPLEMENTATION

All Cray Research systems except CRAY J90 series and CRAY EL series

DESCRIPTION

The `ssread` system call moves data from a secondary data area reserved with the `ssbreak`(2) system call to a buffer. The `sswrite` system call moves data from a buffer to a secondary data area.

The `ssread` system call accepts the following arguments:

pds Specifies a word-aligned address of a buffer.

sds Specifies the secondary data area offset. This is a 4096-byte sector offset in the process' secondary data area. It is specified numerically, with 0 giving the beginning block, 1 giving the second block, and so on. It must be allocated with `ssbreak`(2).

count Specifies the number of 4096-byte blocks to be moved.

CAUTIONS

The `ssread` and `sswrite` system calls work only on a system with an SSD solid-state storage device in its configuration, and a secondary data segment area (SDS) must be configured as a slice of the SSD.

RETURN VALUES

If `ssread` or `sswrite` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `ssread` and `sswrite` system calls fail if one of the following error conditions occurs:

Error Code	Description
`EFAULT`	The request exceeds the boundaries of either the buffer or the secondary data area.
`EIO`	An error occurred during the data transfer.

FORTRAN EXTENSIONS

The `ssread` system call can be called from Fortran as a function:

```
INTEGER pds (512*n), sds, words, SSREAD, I
I = SSREAD (pds, sds, words)
```

The third argument to the Fortran interface to `SSREAD` specifies the number of words to be read. This is different than the third argument to the system call. The `sswrite` system call can be called from Fortran as a function:

```
INTEGER pds (512*n), sds, words, SSWRITE, I
I = SSWRITE (pds, sds, words)
```

The third argument to the Fortran interface to `SSWRITE` specifies the number of words to be written. This is different than the third argument to the system call.

EXAMPLES

The following example shows how to use the `ssread` system call in conjunction with other system calls to transfer data to and from the SDS allocation for a process. In this portion of the program, the `ssbreak`(2) request asks for an SDS area of 1000 blocks, and the `sswrite` request then transfers 1000 blocks of data from the user's process memory space to the process's SDS allocation. Lastly, `ssread` reads the 1000 blocks of data back into the user's process memory from the SDS allocation.

```
int size;
char buff[4096 * 1000];

if ((size = ssbreak(1000L)) == -1) {
  perror("sds allocation error");
  exit(1);
}
else {
  printf("The size of the sds is now %d - 4096-byte blocks\n", size);
}

/* SDS write illustration */

if (sswrite( (long) buff, 0L, 1000L) == -1) {
  perror("sswrite error");
  exit(2);
}

/* SDS read illustration */

if (ssread( (long) buff, 0L, 1000L) == -1) {
  perror("ssread error");
  exit(3);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `ssread` and `sswrite` system calls

SEE ALSO

`ssbreak`(2)

`assign`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ssd`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`stat`, `lstat`, `fstat` – Gets file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *path, struct stat *buf);
int lstat (const char *path, struct stat *buf);
int fstat (int fildes, struct stat *buf);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `stat`, `fstat`)

DESCRIPTION

The `stat` system call obtains information about the file specified by *path*. All directories in the path name leading to the file must be searchable, although it is not necessary to have read, write, or execute permission to the file.

The `lstat` system call is similar to `stat`, except when the specified file is a symbolic link. In this case, `lstat` returns information about the link, and `stat` returns information about the file that the link references.

The `fstat` system call obtains the same information as `stat` about a specified open file. When using `fstat` on a file descriptor returned from the `accept`(2), `socket`(2), or `socketpair`(2) system call, only the `st_uid`, `st_gid`, `st_slevel`, `st_blksize`, `st_oblksize`, and the type portion of `st_mode` fields are meaningful. All other fields will be 0.

The `stat`, `lstat`, and `fstat` system calls accept the following arguments:

path Points to a file's path name (`stat` and `lstat` only). Read, write, or execute permission of the specified file is not required, but all directories listed in the path name leading to the file must be searchable.

buf Points to the `stat` structure.

fildes Specifies a file descriptor. It is obtained from a successful `accept`(2), `creat`(2), `dup`(2), `fcntl`(2), `open`(2), `pipe`(2), `socket`(2), or `socketpair`(2) system call (`fstat` only).

A `stat` structure includes the following members:

```
mode_t    st_mode;      /* File mode; see mknod(2). */
ino_t     st_ino;       /* Inode number for this file */
dev_t     st_dev;       /* Device on which this file resides */
dev_t     st_rdev;      /* Device ID; this entry is defined only */
                        /* for character special or block special files. */
nlink_t   st_nlink;     /* Number of links */
uid_t     st_uid;       /* User ID of the file’s owner */
gid_t     st_gid;       /* Group ID of the file’s group */
int       st_acid;      /* Account id of the file */
off_t     st_size;      /* File size in bytes */
time_t    st_atime;     /* Time that file data was last accessed; */
                        /* changed by system calls creat(2), mknod(2), */
                        /* pipe(2), utime(2), and read(2). */
time_t    st_mtime;     /* Time when data was last modified; */
                        /* changed by system calls creat(2), mknod(2), */
                        /* pipe (2), utime(2), and write(2). */
time_t    st_ctime;     /* Time when file status was last changed; */
                        /* times measured in seconds since 00:00:00 */
                        /* GMT, January 1, 1970.  Changed by */
                        /* system calls chmod(2), chown(2), creat(2), */
                        /* link(2), mknod(2), pipe(2), unlink(2), utime(2), */
                        /* and write(2). */
int       st_count;     /* Reference count from inode; number of active */
                        /* file table entries */
int       st_blocks;    /* Number of 4096 byte blocks allocated to the file */
unsigned int
          st_msref:1,   /* Modification signature referenced flag */
          st_ms:31,     /* Modification signature */
          st_gen;       /* Inode generation number */
int       st_param[8];  /* Device parameter words; this entry is defined only */
                        /* for character special or block special files */
ushort    st_dm_mode;   /* Actual file mode when migrated */
long      st_dm_status; /* Migrated file status flags */
long      st_dm_mid;    /* Migrated file machine id */
long      st_dm_key;    /* Migrated file key */
unsigned int
          st_hasacl:l,  /* File has an ACL */
          st_hascomps:l;/* File has compartments */

short     st_slevel;    /* File level */
short     st_secflg;    /* Security flags (not used) */
short     st_intcls;    /* Integrity class (not used) */
short     st_intcat;    /* Integrity category (not used) */

long      st_site;      /* Site field from inode */
long      st_allocf;    /* Allocation control flags; see fcntl(2) */
```

The file type S_IFREG is returned in st_mode if an IFOFL (offline file) is encountered. The actual file type S_IFOFL is returned in st_dm_mode.

NOTES

The process must have read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component (stat/lstat system calls only).

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_DAC_OVERRIDE	The process is granted search permission to a component of the path prefix via the permission bits and access control list (stat/lstat system calls only).
PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label (stat/lstat system calls only).
PRIV_MAC_READ	The process is granted read permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix (stat/lstat system calls only) and is granted read permission to the file via the security label.

RETURN VALUES

If stat, lstat, or fstat completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The stat or lstat system call fails if one of the following error conditions occurs:

Error Code	**Description**
EACCES	Search permission is denied for a component of the path prefix.
EACCES	The process is not granted read permission to the file via the security label, and the process does not have appropriate privilege.
EFAULT	The *buf* or *path* argument points to an address that is not valid.
ENAMETOOLONG	The supplied file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.

The `fstat` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EBADF`	The *fildes* argument is not a valid open file descriptor.
`EBADF`	The process is not granted read permission to the file via the security label, and the process does not have appropriate privilege.
`EFAULT`	The *buf* argument points to an address that is not valid.

FORTRAN EXTENSIONS

The `stat` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER buf(m), STAT, I
I = STAT (path, buf)
```

Alternatively, `stat` can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
INTEGER buf(m)
CALL STAT (path, buf)
```

The Fortran program must not specify both the subroutine call and the function reference to `stat` from the same procedure. *path* may also be an integer variable. In this case, it must be packed 8 characters per word and terminated with a null (0) byte. The `PXFSTAT`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

The following examples illustrate how to use the `stat` and `lstat` system calls.

Example 1: This example shows the simplest form of the `stat` system call. The following `stat` request provides status information for a file whose path name is supplied as an argument.

```
#include <sys/types.h>
#include <sys/stat.h>

main(int argc, char *argv[])
{
     struct stat buf;

     if (stat(argv[1], &buf) == -1) {
          perror("stat failed");
          exit(1);
     }

     /* Data from the specified file’s inode now available in buf. */
}
```

Example 2: This example shows how to use the `lstat`, `readlink`(2), and `stat` requests. It uses the list of file names supplied as arguments to produce a display listing each file name along with the size of each file. If any file in the argument list is a symbolic link, the program also displays the path name of the file that is the target of the link as well as that file's size. For a definition of `S_IFLNK`, see the `sys/stat.h` file.

```
#include <sys/types.h>
#include <sys/stat.h>

main(int argc, char *argv[])
{
     char file[50], tfile[50];
     struct stat buf;
     int i;

     for (i = 1; i < argc; i++) {
          strcpy(file, argv[i]);
          if (lstat(argv[i], &buf) == -1) {
               perror("lstat failed");
               continue;
          }
          if ((buf.st_mode & S_IFLNK) == S_IFLNK) {    /* a symbolic link? */
               readlink(argv[i], tfile, 50);
               if (stat(tfile, &buf) == -1) {
                    perror("stat failed");
                    exit(1);
               }
               strcat(file, "->");
               strcat(file, tfile);
          }
          printf("%-50s   %d\n", file, buf.st_size);
     }
}
```

SEE ALSO

`accept`(2), `chmod`(2), `chown`(2), `creat`(2), `dmofrq`(2), `dup`(2), `fcntl`(2), `link`(2), `mknod`(2), `open`(2), `pipe`(2), `readlink`(2), `socket`(2), `socketpair`(2), `time`(2), `unlink`(2)

`PXFSTAT`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`statfs`, `fstatfs` – Gets file system information

SYNOPSIS

`#include <sys/statfs.h>`

`int statfs (char *`*path*`, struct statfs *`*buf*`, int` *len*`, int` *fstyp*`);`

`int fstatfs (int` *fildes*`, struct statfs *`*buf*`, int` *len*`, int` *fstyp*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `statfs` system call returns a *generic superblock* describing a file system. It can be used to acquire information about mounted and unmounted file systems, and usage is slightly different in the two cases. The `statfs` and `fstatfs` system calls accept the following arguments:

path Specifies path to a file system.

buf Points to a structure. It will be filled by the system call, as described in the following text.

len Specifies the number of bytes of information that the system should return in the structure.

The *len* argument must be no greater than `sizeof` (`struct statfs`), and ordinarily it contains exactly that value; if it holds a smaller value, the system fills the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

fstyp Specifies file system type.

fildes Specifies open file descriptor.

If the file system of interest is currently mounted, *path* must specify a file that resides on that file system. In this case, the file system type is known to the operating system, and the *fstyp* argument must be 0. For an unmounted file system, *path* must specify the block special file containing it, and *fstyp* must contain the nonzero file system type. In both cases, read, write, or execute permission of the specified file is not required, but all directories listed in the path name leading to the file must be searchable.

The `statfs` structure to which *buf* points includes the following members:

```
short   f_fstyp;       /* File system type */
long    f_bsize;       /* Block size */
long    f_frsize;      /* Fragment size */
long    f_blocks;      /* Total number of blocks */
long    f_bfree;       /* Count of free blocks */
long    f_files;       /* Total number of file nodes */
long    f_ffree;       /* Count of free file nodes */
char    f_fname[6];    /* Volume name */
char    f_fpack[6];    /* Pack name */
long    f_priparts;    /* Bitmap of primary partitions */
long    f_secparts;    /* Bitmap of secondary partitions */
long    f_npart;       /* Number of partitions (logical drives) in FS*/
long    f_bigsize;     /* Block to "bigunit" allocation crossover */
long    f_bigunit;     /* Allocation size for large files */
long    f_prinblks;    /* Total number of 512 wd blocks in primary */
long    f_prinfree;    /* Number of free 512 wd blocks in primary */
long    f_priaunit;    /* Size of primary area allocation unit */
long    f_secnblks;    /* Total number of 512 wd blocks in secondary */
long    f_secnfree;    /* Number of free 512 wd blocks in secondary */
long    f_secaunit;    /* Size of secondary area allocation unit */
```

The `fstatfs` system call is similar, except that the file specified by *path* in `statfs` is identified instead by an open file descriptor, *fildes*, obtained from a successful `creat`(2), `dup`(2), `fcntl`(2), `open`(2), or `pipe`(2) system call.

If *fstyp* indicates the network file system (NFS) file system, the fields of the structure have the following meanings:

`f_frsize`	0
`f_bsize`	Block size on a remote system
`f_blocks`	Number of blocks on remote file system
`f_bfree`	Free blocks on remote file system
`f_files`	0
`f_ffree`	0
`f_fname`	Name of host in which remote file system resides

NOTES

The `statfs` system call obsoletes the `ustat`(2) system call for most purposes.

To be granted search permission to a component of the path prefix (for the `statfs` system call), the active security label of the process must be greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to a component of the path prefix via the permission bits and access control list (`statfs` system call only).
`PRIV_MAC_READ`	The process is granted search permission to a component of the path prefix via the security label (`statfs` system call only).

If the `PRIV_SU` configuration option is enabled, the super user is granted search permission to every component of the path prefix.

RETURN VALUES

If `statfs` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `statfs` or the `fstatfs` system call fails if one of the following error conditions occurs:

Error Code	Description
`EACCES`	Search permission is denied for a component of the path prefix.
`EBADF`	The *fildes* argument is not a valid open file descriptor.
`EBADF`	The calling process does not have MAC read access to the file to which the file descriptor refers.
`EFAULT`	The *buf* or *path* argument points to an address that is not valid.
`EINVAL`	The *fstyp* argument is an not a valid file system type; the *path* argument is not a block special file, and the *fstyp* argument is nonzero; the *len* argument is negative or is greater than `sizeof` (`struct statfs`).
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.

EXAMPLES

This example shows how to use the `statfs` and `sysfs` system calls to obtain file system information. The `statfs` request retrieves information for the file system containing the file whose name is passed as an argument. The `sysfs` system call converts the numerical file system type to a character-string format before displaying it.

```
#include <sys/types.h>
#include <sys/statfs.h>
#include <sys/fstyp.h>
#include <sys/fsid.h>

main(int argc, char *argv[])
{
     struct statfs stats;
     char buf[FSTYPSZ];

     if (statfs(argv[1], &stats, sizeof(struct statfs), 0) == -1) {
          perror("statfs error");
          exit(1);
     }

     if (sysfs(GETFSTYP, stats.f_fstyp, buf) == -1) {
          perror("sysfs (GETFSTYP) error");
          exit(1);
     }

     printf("File system type = %s\n", buf);
     printf("Block size = %d\n", stats.f_bsize);
     printf("Fragment size = %d\n", stats.f_frsize);
     printf("Total number of blocks on file system = %d\n", stats.f_blocks);
     printf("Total number of free blocks = %d\n", stats.f_bfree);
     printf("Total number of file nodes (inodes) = %d\n", stats.f_files);
     printf("Total number of free file nodes = %d\n", stats.f_ffree);
     printf("Volume name = %s\n", stats.f_fname);
     printf("Pack name = %s\n", stats.f_fpack);
     printf("Primary partition bit map = %o\n", stats.f_priparts);
     printf("Secondary partition bit map = %o\n", stats.f_secparts);
     printf("Number of partitions = %d\n", stats.f_npart);
     printf("Big file threshold = %d bytes ", stats.f_bigsize);
     printf("or %d blocks\n", stats.f_bigsize/stats.f_bsize);
     printf("Big file allocation unit size = %d bytes ", stats.f_bigunit);
     printf("or %d blocks\n", stats.f_bigunit/stats.f_bsize);
     printf("Number of blocks in primary partitions = %d\n", stats.f_prinblks);
     printf("Number of free blocks in primary partitions = %d\n",
                                                          stats.f_prinfree);
     printf("Primary partition allocation unit size = %d ", stats.f_priaunit);
     printf("bytes or %d blocks\n", stats.f_priaunit/stats.f_bsize);
     printf("Number of blocks in secondary partitions = %d\n",stats.f_secnblks);
     printf("Number of free blocks in secondary partitions = %d\n",
                                                          stats.f_secnfree);
     printf("Secondary partition allocation unit size = %d ",stats.f_secaunit);
     printf("bytes or %d blocks\n", stats.f_secaunit/stats.f_bsize);
}
```

SEE ALSO

`chmod`(2), `chown`(2), `creat`(2), `dup`(2), `fcntl`(2), `link`(2), `mknod`(2), `open`(2), `pipe`(2), `read`(2), `time`(2), `unlink`(2), `ustat`(2), `utime`(2), `write`(2)

`fs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`statvfs`, `fstatvfs` – Gets file system information

SYNOPSIS

```
#include <sys/statvfs.h>
int statvfs (const char *path, struct statvfs *buf);
int fstatvfs (int fildes, struct statvfs *buf);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `statvfs` system call obtains information about the file specified by *path*. All directories in the path name leading to the file must be searchable, although it is not necessary to have read, write, or execute permission to the file.

The `fstatvfs` system call obtains the same information as `statvfs` about the file referenced by *fildes*

The `statvfs` and `fstatvfs` system calls accept the following arguments:

path Points to a file's path name (`statvfs` only).

buf Points to the `statvfs` structure.

fildes Specifies a file descriptor (`fstatvfs` only).

The following flags can be returned in the `f_flag` member:

ST_RDONLY read-only file system

ST_NOSUID setuid/setgid bits ignored by exec

ST_NOTRUNC does not truncate long file names

NOTES

The process must have read permission to the file via the security label. That is, the active security label of the process must be greater than or equal to the security label of the file.

To be granted search permission to a component of the path prefix, the active security label of the process must be greater than or equal to the security label of the component (`statvfs` system calls only).

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_DAC_OVERRIDE	The process is granted search permission to a component of the path prefix via the permission bits and access control list (statvfs system calls only).
PRIV_MAC_READ	The process is granted search permission to a component of the path prefix via the security label (statvfs system calls only).
PRIV_MAC_READ	The process is granted read permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix (statvfs system calls only) and is granted read permission to the file via the security label.

RETURN VALUES

If statvfs or fstatvfs completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The statvfs or fstatvfs system call fails if one of the following error conditions occurs:

Error Code	Description
EIO	An I/O error occurred while reading the file system.
EINTR	A signal was caught during execution of the function.

The statvfs system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
ELOOP	Too many symbolic links were encountered in resolving the path.
ENAMETOOLONG	The supplied file name is too long.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.

The fstatvfs system call fails if the following error conditions occurs:

Error Code	Description
EBADF	The *fildes* argument is not a valid open file descriptor.
EBADF	The process is not granted read permission to the file via the security label, and the process does not have appropriate privilege.

FILES

sys/statvfs.h

SEE ALSO

chmod(2), chown(2), creat(2), dup(2), exec(2), fcntl(2), link(2), mknod(2), open(2), pipe(2), read(2), time(2), unlink(2) utime(2) write(2)

NAME

stime – Sets time

SYNOPSIS

```
#include <unistd.h>
int stime (long *tp);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The stime system call sets the system time and date.

tp Points to the value of time as measured in seconds from 00:00:00 Greenwich mean time (GMT), January 1, 1970. Only a process with appropriate privilege can use this system call.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_TIME	The process is allowed to use this system call.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_SYSPARAM permbit is allowed to use this system call.

RETURN VALUES

If stime completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The stime system call fails if the following error condition occurs:

Error Code	**Description**
EPERM	The process did not have appropriate privilege to use this system call.

FILES

/usr/include/unistd.h	Contains C prototype for the stime system call

SEE ALSO

time(2)

NAME

suspend, resume – Controls execution of processes

SYNOPSIS

```
#include <sys/category.h>
#include <unistd.h>
```

int suspend (int *category*, int *id*);

int resume (int *category*, int *id*);

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The suspend system call makes a process or group of processes ineligible to execute; the resume system call restores a process or group of processes to be eligible to execute.

The suspend and resume system calls accept the following arguments:

category Specifies one of the following categories: C_PROC, C_PGRP, or C_SESS

id Specifies the PID, PGRP, or SID corresponding to the *category*. A PID of 0 means that the current process is affected, and a PID of −1 means that all processes except the current process are affected. Similarly, a PGRP of 0 means that all processes in the current process group are affected, and a PGRP of −1 means that all processes not in the current process group are affected. A SID of 0 means that all processes in the current session are affected. System processes, such as processes 0 and 1, are never suspended.

The calling process must be the owner of the specified process or have appropriate privilege. If an affected process is not part of the calling process' session, the calling process must have appropriate privilege.

NOTES

The active security label of the calling process must be equal to the active security label of every affected process.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
PRIV_ADMIN	The calling process is allowed to suspend or resume a process that is not part of its session.
PRIV_MAC_WRITE	The calling process is allowed to override the security label restrictions for suspend and resume.
PRIV_POWNER	The calling process is considered the owner of the specified process.

If the `PRIV_SU` configuration option is enabled, the super user is considered the owner of every affected process and is allowed to suspend or resume a process that is not part of its session. If the `PRIV_SU` configuration option is enabled, the super user is allowed to override security label restrictions.

RETURN VALUES

If `suspend` or `resume` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `suspend` or `resume` system call fails if one of the following error conditions occurs:

Error Code	Description
`EAGAIN`	One of the processes being suspended was never in a suspendible state during the last 120 seconds. The `suspend` system call may be attempted again.
`EINTR`	An asynchronous signal (such as `interrupt` or `quit`), which you have elected to catch, occurred during a `suspend` system call. When execution resumed after processing the signal, the interrupted system call returned this error condition.
`EINVAL`	One of the arguments contains a value that is not valid.
`EPERM`	The calling process does not own an affected process and does not have appropriate privilege.
`EPERM`	The calling process is attempting to suspend or resume a process that is not part of its session and does not have appropriate privilege.
`ESRCH`	No process can be found that matches the *category* and *id* requests.
`ESRCH`	The calling process does not meet security label requirements and does not have appropriate privilege.
`ESRCH`	The calling process does not own any processes in the requested process group or session and does not have appropriate privilege.

FORTRAN EXTENSIONS

The `suspend` system call can be called from Fortran as a function:

```
INTEGER category, id, SUSPEND, I
I = SUSPEND (category, id)
```

The `resume` system call can be called from Fortran as a function:

```
INTEGER category, id, RESUME, I
I = RESUME (category, id)
```

EXAMPLES

The following examples show how to use the `suspend` and `resume` system calls to suspend and resume program execution.

Example 1: This program suspends itself using the `suspend` request. When the process resumes, it computes the number of seconds it was in a suspended state.

```
#include <sys/category.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

main()
{
     time_t stime, etime;

     stime = time((long *) 0);

     if (suspend(C_PROC, 0) == -1) {
          perror("suspend failed");
          exit(1);
     }

     etime = time((long *) 0);
     printf("Program was suspended for %ld seconds\n", etime - stime);
}
```

Example 2: Using the `resume` system call, this program resumes any suspended process whose process identification number (PID) is supplied as an argument.

```
#include <sys/category.h>
#include <stdio.h>
#include <unistd.h>

main(int argc, char *argv[])
{
     int pid;

     sscanf(argv[1], "%d", &pid);  /* convert pid string to int */

     if (resume(C_PROC, pid) == -1) {
          perror("resume failed");
          exit(1);
     }
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `suspend` and `resume` system calls

SEE ALSO

`suspend`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`symlink` – Makes a symbolic link to a file

SYNOPSIS

```
#include <unistd.h>
int symlink (char *name1, char *name2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `symlink` system call makes a symbolic link to a file. It accepts the following arguments:

name1 Specifies the string used in creating the symbolic link.

name2 Specifies the name of the file created.

A symbolic link *name2* is created to *name1*. Either name may be an arbitrary path name; the files do not need to be on the same file system.

NOTES

The active security label of the calling process must fall within the security label range of the file system on which *name2* will reside.

To be granted search permission to a component of the path prefix of *name2*, the active security label of the process must be greater than or equal to the security label of the component.

To be granted write permission to the parent directory of *name2*, the active security label of the process must be equal to the security label of the directory.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the *name2* path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the parent directory of *name2* via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to every component of the *name2* path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the parent directory of *name2* via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted search permission to every component of the path prefix and is granted write permission to the parent directory of *name2*.

RETURN VALUES

If symlink completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The symlink system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix of *name2*.
EACCES	Write permission is denied to the parent directory of *name2*.
EEXIST	The file referred to by *name2* already exists.
EFAULT	The *name1* or *name2* argument points outside the process allocated address space.
EFLNEQ	The active security label of the calling process does not fall within the range of the file system on which *name2* will reside.
EINVAL	The *name2* argument contains a character with the high-order bit set.
EIO	An I/O error occurred during a read from or write to the file system.
EMLINK	Too many symbolic links were encountered in translating *name2*.
ENAMETOOLONG	A component of either *name1* or *name2* exceeds 255 characters, or either *name1* or *name2* exceeds 1023 characters.
ENOENT	A component of the path prefix of *name2* does not exist.
ENOSPC	The directory in which the entry for the new symbolic link is being placed cannot be extended because of one of the following: • There is no space left in the file system to make the directory longer. Sometimes, but not always, the new directory name added by symlink requires that an additional block be allocated. • There was not enough space (one block) to write the *name2* string to disk.
ENOSPC	The new symbolic link cannot be created because no space left is on the file system that will contain the link.
ENOSPC	No free inodes exist on the file system on which the file is being created.
ENOTDIR	A component of the path prefix of *name2* is not a directory.

`EQACT`	The new symbolic link cannot be created for one of the following reasons: the quota of inodes on the file system on which the file is being created has been exhausted for the current account, the account's quota of disk blocks on the file system that will contain the link has been exhausted, or the directory in which the entry for the new symbolic link is being placed cannot be extended because the account's quota of disk blocks on the file system containing the directory has been exhausted.
`EQGRP`	The new symbolic link cannot be created for one of the following reasons: the quota of inodes on the file system on which the file is being created has been exhausted for the current group, the group's quota of disk blocks on the file system that will contain the link has been exhausted, or the directory in which the entry for the new symbolic link is being placed cannot be extended because the group's quota of disk blocks on the file system containing the directory has been exhausted.
`EQUSR`	The new symbolic link cannot be created for one of the following reasons: the quota of inodes on the file system on which the file is being created has been exhausted for the current user, the user's quota of disk blocks on the file system that will contain the link has been exhausted, or the directory in which the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
`EROFS`	The file *name2* would reside on a read-only file system.

EXAMPLES

This example shows how to use the `symlink` system call to create a symbolic link. The following `symlink` request makes a symbolic link to a file from information supplied as arguments. The first argument, `argv[1]`, is the path name of an existing file or directory that is the target of the new link. The second argument, `argv[2]`, is the name of the new link. The program later forces an `ls -l` display of the new link.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

main(int argc, char *argv[])
{
     static char cmd[50] = {"ls -l "};

     if (argc < 3) {
          fprintf(stderr, "Insufficient arguments supplied!\n");
          exit(1);
     }

     if (symlink(argv[1], argv[2]) == -1) {
          perror("symlink failed");
          exit(1);
     }

     strcat(cmd, argv[2]);
     system(cmd);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `symlink` system call

SEE ALSO

`link`(2), `lstat`(2), `readlink`(2), `stat`(2), `unlink`(2)

`ln`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`sync` – Flushes system buffers out of main memory

SYNOPSIS

```
#include <unistd.h>
void sync (void);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `sync` system call causes all information in memory that should be on disk to be flushed out of main memory, including modified inodes and delayed block I/O.

Information is flushed to the logical device cache (a second-level cache) or disk if a logical device cache is not configured. The `ldsync`(8) command flushes data from the logical device cache to disk.

Use `sync` in programs that examine a file system (for example, `fsck`(8) and `df`(1)). You must execute `sync` before halting or rebooting the system.

The `sync` system call issues the write request; it may return before all of the data is written.

RETURN VALUES

None

FORTRAN EXTENSIONS

The `sync` system call can be called from Fortran as a function:

```
INTEGER SYNC, I
I = SYNC ( )
```

Alternatively, `sync` can be called from Fortran as a subroutine, because there is no return value:

```
CALL SYNC ( )
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `sync` system call

SEE ALSO

`fsync`(2), `ioctl`(2)

`df`(1), `sync`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`fsck`(8), `ldsync`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`sysconf` – Retrieves system implementation information

SYNOPSIS

```
#include <unistd.h>
long sysconf (int name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `sysconf` system call provides a method for an application to determine the current value of a configurable system limit or option. It accepts the following argument:

name Represents the system variable to be queried.

The values for *name* specified by the POSIX P1003.1 standard are listed in the following with a brief description of the value each returns:

`_SC_ARG_MAX`	The maximum length of arguments in bytes for `exec`( ).
`_SC_CHILD_MAX`	The maximum number of processes allowed per user.
`_SC_CLK_TCK`	The number of clock ticks per second.
`_SC_JOB_CONTROL`	The POSIX job control option has been implemented; if true, it is 1.
`_SC_NGROUPS_MAX`	The multigroups size; if multigroups are not implemented, it is 0.
`_SC_OPEN_MAX`	The maximum number of open files.
`_SC_PID_MAX`	The maximum value for a process ID. (This name is no longer specified by POSIX P1003.1.)
`_SC_SAVED_IDS`	The `exec`( ) routine saves the real UID and GID of the caller for later use; if true, it is 1.
`_SC_STREAM_MAX`	The number of streams that one process can have open at any given time.
`_SC_TZNAME_MAX`	The maximum number of bytes supported for the name of a time zone.
`_SC_UID_MAX`	The maximum value for a user ID. (This name is no longer specified by POSIX P1003.1.)
`_SC_VERSION`	The version/revision of the POSIX standard used for this implementation.

The values for *name* specified by the POSIX P1003.2 standard are listed in the following with a brief description of the value each returns:

`_SC_BC_BASE_MAX`	The maximum *obase* value allowed by the `bc`(1) utility.
`_SC_BC_DIM_MAX`	The maximum number of elements permitted in an array by the `bc`(1) utility.
`_SC_BC_SCALE_MAX`	The maximum *scale* value allowed by the `bc`(1) utility.
`_SC_BC_STRING_MAX`	The maximum length of a string constant accepted by the `bc`(1) utility.
`_SC_COLL_WEIGHTS_MAX`	The maximum number of weights that can be assigned to an entry of the `LC_COLLATE` *order* keyword in the locale definition file.
`_SC_EXPR_NEST_MAX`	The maximum number of expressions that can be nested within parentheses by the `expr`(1) utility.
`_SC_LINE_MAX`	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing newline character.
`_SC_RE_DUP_MAX`	The maximum number of repeated occurrences of a regular expression permitted when using the interval notation *m,n*.
`_SC_2_VERSION`	The C-language development facilities support the POSIX P1003.2 C-Language Bindings Option.
`_SC_2_C_DEV`	The system supports the POSIX P1003.2 C-Language Development Utilities Option.
`_SC_2_FORT_DEV`	The system supports the POSIX P1003.2 FORTRAN Development Utilities Option.
`_SC_2_FORT_RUN`	The system supports the POSIX P1003.2 FORTRAN Runtime Utilities Option.
`_SC_2_LOCALEDEF`	The system supports the POSIX P1003.2 Locale Creation Option.
`_SC_2_SW_DEV`	The system supports the POSIX P1003.2 Software Development Utilities Option.
`_SC_2_C_BIND`	The system supports the POSIX P1003.2 C-Language Bindings Option.
`_SC_2_CHAR_TERM`	The system supports at least one terminal type capable of all operations in the POSIX standard.
`_SC_2_C_VERSION`	The version of the POSIX P1003.2 interfaces used for this implementation.

The values for *name* specified by the X/Open XPG4 standard are listed in the following with a brief description of the value each returns:

`_SC_PASS_MAX`	The maximum size of a password.
`_SC_XOPEN_VERSION`	The version of the X/Open standard supported by this implementation.

`_SC_XOPEN_CRYPT`	The system supports the X/Open Encryption Feature Group.
`_SC_XOPEN_ENH_I18N`	The system supports the X/Open Enhanced Internationalization Feature Group.
`_SC_XOPEN_SHM`	The system supports the X/Open Shared Memory Feature Group.
`_SC_LOGIN_NAME_MAX`	The maximum length of a login name.
`_SC_TTY_NAME_MAX`	The maximum length of a tty path name.
`_SC_GETGR_R_SIZE_MAX`	The maximum size of data buffers used by the `getgrgid_r` and `getgrnam_r` library functions.
`_SC_GETPW_R_SIZE_MAX`	The maximum size of data buffers used by the `getpwgid_r` and `getpwnam_r` library functions.

The values for *name* that are unique to Cray Research are listed in the following with a brief description of the value each returns. These unique values will not change.

`_SC_CRAY_AVL`	Additional vector logical hardware; if present, it is 1.
`_SC_CRAY_BDM`	Bidirectional memory enabled; if true, it is 1.
`_SC_CRAY_BMM`	Bit matrix multiply unit; if present, it is 1.
`_SC_CRAY_CHIPSZ`	The memory chip size.
`_SC_CRAY_CPCYCLE`	The CPU cycle time in picoseconds.
`_SC_CRAY_EMA`	Extended memory addressing hardware; if present, it is 1.
`_SC_CRAY_HPM`	Hardware performance monitor hardware; if present, it is 1.
`_SC_CRAY_IOS`	The I/O subsystem type; `IOS_MODEL_E`.
`_SC_CRAY_MFSUBTYPE`	The mainframe subtype (see `sys/sn.h` and `sys/machd.h`).
`_SC_CRAY_MFTYPE`	The mainframe type (see `sys/sn.h` and `sys/machd.h` for all systems).
`_SC_CRAY_NBANKS`	The number of memory banks on the Cray Research mainframe.
`_SC_CRAY_NBUF`	Number of 512-word system I/O cache blocks.
`_SC_CRAY_NCPU`	The number of CPUs currently available.
`_SC_CRAY_NDISK`	The number of disk devices configured on the system.
`_SC_CRAY_NMOUNT`	The number of file-system mount points configured in the system.
`_SC_CRAY_NPTY`	The maximum number of pty devices configured into the currently running version of the operating system.
`_SC_CRAY_NUSERS`	The number of users configured.
`_SC_CRAY_NVHISP`	The number of VHISP channels to the SSD solid-state storage device.
`_SC_CRAY_OPEN_MAX`	The value of the largest open file limit supported by the kernel.

`_SC_CRAY_OS_HZ`	The frequency per second (usually 100) with which the operating system clock routine is called.
`_SC_CRAY_RELEASE`	The release level of the currently running version of the operating system. The release level is multiplied by 1000 (for example, release level 5.0 = 5000, release level 5.1 = 5100, and so on).
`_SC_CRAY_SCTRACE`	System call timing; if on, it is 1.
`_SC_CRAY_SDS`	Size of secondary data segment (SDS) memory in 512-word blocks.
`_SC_CRAY_SECURE_MAC`	The system supports `syshigh` and `syslow` security labels. This implies that file systems have been appropriately labeled.
`_SC_CRAY_SECURE_SYS`	The system has been generated with security enabled. Always returns TRUE (nonzero).
`_SC_CRAY_SERIAL`	The system serial number (see `sys/sn.h`).
`_SC_CRAY_SSD`	Size of the SSD in words.
`_SC_CRAY_SYSMEM`	The size of the kernel and tables, in words.
`_SC_CRAY_USRMEM`	The user memory available, in words.

RETURN VALUES

If *name* not a valid value, the `sysconf` system call returns a value of –1, and sets `errno` to `EINVAL`. If *name* is valid, `sysconf` returns the current variable value for the system.

EXAMPLES

This example shows how to use the `sysconf` system call to retrieve system implementation information. The following `sysconf` requests illustrate some of the different types of information available through this call. Because `sysconf` returns the mainframe type as an integer, the programmer creates a table to convert the mainframe type to a more recognizable character string.

```
#include <unistd.h>

main()
{
     /* The following table based upon the mainframe definitions
        in the sys/machd.h and sys/machcons.h header files. */
     static char *mftype[] = {"", "CRAY Y-MP", "", "CRAY C90"};

     printf("The mainframe type = %s\n",
             mftype[sysconf(_SC_CRAY_MFTYPE)]);
     printf("The current number of available cpu’s = %ld\n",
             sysconf(_SC_CRAY_NCPU));
     printf("The size of the kernel and tables = %ld words\n",
             sysconf(_SC_CRAY_SYSMEM));
     printf("The amount of user memory available = %ld words\n",
             sysconf(_SC_CRAY_USRMEM));
     printf("The number of clock ticks per second = %ld\n",
             sysconf(_SC_CLK_TCK));
}
```

FILES

`/usr/include/sys/machd.h`	Contains machine-dependent information
`/usr/include/sys/sn.h`	Contains Cray Research mainframe hardware information
`/usr/include/sys/tfm.h`	Defines `TFM_UDB_6`
`/usr/include/unistd.h`	Contains C prototype for the `sysconf` system call

SEE ALSO

`pathconf`(2)

`bc`(1), `expr`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`sysfs` – Gets file system type information

SYNOPSIS

```
#include <sys/fstyp.h>
#include <sys/fsid.h>
```

`int sysfs (int` *opcode*`, char *`*fsname*`);`

`int sysfs (int` *opcode*`, int` *fs_index*`, char *`*buf*`);`

`int sysfs (int` *opcode*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `sysfs` system call returns information about the file system types configured in the system. The number of arguments accepted by `sysfs` varies and depends on the *opcode*.

opcode Specifies function to perform. The following are valid *opcode* values:

`GETFSIND` Translates *fsname*, a null-terminated file system identifier, into a file system type index.

`GETFSTYP` Translates *fs_index*, a file system type index, into a null-terminated file system identifier and writes it into the buffer to which *buf* points. This buffer must be at least of size `FSTYPSZ`, as defined in `sys/fstyp.h`.

`GETNFSTYP` Returns the total number of file system types configured in the system.

fsname Specifies file system identifier.

fs_index Specifies file system type index.

buf Points to a buffer.

RETURN VALUES

If `sysfs` completes successfully, it returns the file system type index if *opcode* is `GETFSIND`, a value of 0 if *opcode* is `GETFSTYP`, or the number of file system types configured if *opcode* is `GETNFSTYP`. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `sysfs` system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The *buf* or *fsname* argument points to a user address that is not valid.
EINVAL	The *fsname* argument points to a file system identifier that is not valid; *fs_index* is 0, or not valid; *opcode* is not valid.

EXAMPLES

This example shows how to use the `sysfs` and `statfs`(2) system calls to obtain file system information. The `statfs`(2) request retrieves information for the file system containing the file whose name is passed as an argument. The `sysfs` system call converts the numerical file system type to a character-string format before displaying it. The final `sysfs` request determines the total number of file system types configured in the system.

```
#include <sys/types.h>
#include <sys/statfs.h>
#include <sys/fstyp.h>
#include <sys/fsid.h>

main(int argc, char *argv[])
{
     struct statfs stats;
     char buf[FSTYPSZ];
     int nconfig;

     if (statfs(argv[1], &stats, sizeof(struct statfs), 0) == -1) {
          perror("statfs error");
          exit(1);
     }

     if (sysfs(GETFSTYP, stats.f_fstyp, buf) == -1) {
          perror("sysfs (GETFSTYP) error");
          exit(1);
     }

     printf("File system type => %s\n", buf);
     printf("Block size = %d\n", stats.f_bsize);
     printf("Fragment size = %d\n", stats.f_frsize);
     printf("Total number of blocks on file system = %d\n", stats.f_blocks);
     printf("Total number of free blocks = %d\n", stats.f_bfree);
     printf("Total number of file nodes (inodes) = %d\n", stats.f_files);
     printf("Total number of free file nodes = %d\n", stats.f_ffree);
     printf("Volume name => %s\n", stats.f_fname);
     printf("Pack name => %s\n\n", stats.f_fpack);
```

```
          if ((nconfig = sysfs(GETNFSTYP)) == -1) {
               perror("sysfs (GETNFSTYP) error");
               exit(1);
          }
          else {
               printf("Number of file system types configured = %d\n", nconfig);
          }
     }
```

SEE ALSO

statfs(2)

NAME

`syssgi` – Provides a system interface to Silicon Graphics workstations

SYNOPSIS

```
#include <sys/syssgi.h>
ptrdiff_t syssgi (int request, ...);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The `syssgi` call is a system interface specific to Silicon Graphics workstations. It accepts the following argument:

request Represents the requested interface. The currently supported values for *request* are listed below.

The following *request* values are interfaces that implement various `libc` functions. They are all subject to change and should not be called directly by applications.

`SGI_GETASH`

`SGI_SETASH`

`SGI_GETPRID`

`SGI_GETDFLTPRID`

`SGI_SETPRID`

`SGI_GETSPINFO`

`SGI_SETSPINFO`

`SGI_NEWARRAYSESS`

The following *request* values are interfaces that implement various `libarray` functions. They are all subject to change and should not be used directly by applications.

`SGI_ENUMASHS`

`SGI_GETARSESS`

`SGI_GETASMACHID`

`SGI_PIDSINASH`

`SGI_SETASMACHID`

RETURN VALUES

If `syssgi` completes successfully, a command-dependent value (default of `0`) is returned; otherwise, a value of `-1` is returned, and `errno` is set to indicate the error.

ERRORS

The `syssgi` system call fails if one of the following conditions occurs:

Error Code	**Description**
`EFAULT`	A buffer is referenced which is not in a valid part of the calling program's address space.
`ENOMEM`	The specified buffer was not large enough to hold the entire list of process IDs returned by the `SGI_PIDSINASH` function.

NAME

`tabinfo`, `tabread` – Returns information on and reads a system table

SYNOPSIS

```
#include <sys/table.h>
int tabinfo (char *name, struct tbs *info);
int tabread (char *name, char *buf, long nbytes, long offset);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `tabinfo` and `tabread` system calls let you read a system table without reading `/dev/kmem`. The `tabinfo` call describes table characteristics: location, header length, number of entries, and size of entry. Using the information returned by `tabinfo`, you can create a user buffer into which `tabread` will read all or part of a table.

If you have read permission on `/dev/kmem`, you will have unlimited access with `tabinfo` and `tabread`, regardless of the table permissions. The calls let you process a table in segments; the requirement for an arbitrarily large buffer does not exist. Using the information from `tabinfo`, you can calculate buffer sizes.

The `tabinfo` and `tabread` system calls accept the following arguments:

name Points to a table name (defined in `sys/table.h`).

info Points to the `tbs` structure to receive the information.

buf Points to the character to which the buffer points to receive the table.

nbytes Specifies the number of bytes to be read.

offset Specifies the number of bytes after the table base at which `tabread` is to start reading.

NOTES

The `tabinfo` and `tabread` system calls are similar to the `nlist`(3C) library routine and have some of the same functionality.

If the process does not have read permission to `/dev/kmem` and the table permissions restrict access, the process must belong to the appropriate table group, or the process must have appropriate privilege.

If the `SECURE_MAC` option is enabled, the calling process uses the `tabread` system call to retrieve process table information and the active security label of the process table entry is greater than the active security label of the calling process, the returned process table entry is zero-filled. A process with appropriate privilege is allowed to override this behavior.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
PRIV_ADMIN	The process is granted access to tables whose permissions restrict access.
PRIV_DAC_OVERRIDE	The process is granted search permission to every directory component of the /dev/kmem path prefix via the permission bits and access control list.
PRIV_DAC_OVERRIDE	The process is granted read permission to /dev/kmem via the permission bits and access control lists.
PRIV_MAC_READ	The process is granted search permission to every directory component of the /dev/kmem path prefix via the security label.
PRIV_MAC_READ	The process is granted read permission to /dev/kmem via the security label.
PRIV_MAC_READ	The process is allowed to read all process table entries. That is, process table entries are not zero-filled.

If the PRIV_SU configuration option is enabled, the super user is allowed to override all tabread and tabinfo restrictions.

RETURN VALUES

If tabinfo or tabread completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The tabinfo or tabread system call fails if one of the following error conditions occurs:

Error Code	Description
EACCES	Access is not permitted.
EFAULT	The address of *info* or *buf* is illegal.
EINVAL	The *name* argument points to an undefined table name.

EXAMPLES

The following example shows how to use the tabinfo and tabread system calls to retrieve information from a system table. In this case, the entire file table from the system is read into the process's memory space.

```
#include <sys/table.h>
#include <stdlib.h>

/* The structure of type tbs defined as follows (from <sys/table.h>):

struct tbs {
        char    name[9];        - ASCII name of table entry -
        long    *addr,          - Start address of table (word *) -
                head,           - Length of table header (chars) -
                ent,            - Number of entries -
                len,            - Length of each entry (chars)-
                perm;           - Permission word -
};  */

main()
{
     struct tbs tinfo;
     char *tloc;
     long tsize;

     if (tabinfo(FILETAB, &tinfo) == -1) {
          perror("tabinfo failed");
          exit(1);
     }

     tsize = tinfo.head + (tinfo.ent * tinfo.len);
     tloc = (char *) malloc(tsize);

     if (tabread(FILETAB, tloc, tsize, 0) == -1) {
          perror("tabread failed");
          exit(1);
     }
}
```

FILES

`/usr/include/sys/table.h` Contains user or system structure declaration

SEE ALSO

`nlist`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

`target` – Retrieves or modifies machine characteristics

SYNOPSIS

```
#include <sys/target.h>
int target (int request, struct target *addr);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `target` system call provides a mechanism for compilers to determine the physical characteristics of the host system.

Users may retrieve machine characteristics for the machine on which they are running (host machine) or for the machine for which they are targeting code (target machine). Only a process with appropriate privilege can modify characteristics for the target machine.

The `target` system call accepts the arguments:

request Specifies the type of request; *request* may be one of the following:

`MC_GET_SYSTEM`	Retrieves the host machine characteristics.
`MC_GET_TARGET`	Retrieves the target machine characteristics.
`MC_SET_TARGET`	Modifies the target machine characteristics (on all systems except Cray MPP systems).

addr Specifies the address of a structure of type `target`.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to modify characteristics of the target machine.

If the `PRIV_SU` configuration option is enabled, the super user or a process with the `PERMBITS_SYSPARAM` permbit is allowed to modify characteristics of the target machine.

RETURN VALUES

If `target` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `target` system call fails if one of the following error conditions occurs:

Error Code	Description
EFAULT	The `target` structure address is not within the user's bounds.
EINVAL	The *request* field is not valid.
EPERM	The process tried to use the `MC_SET_TARGET` request but did not have appropriate privilege.

EXAMPLES

The following example shows how to use the `target` system call to retrieve the characteristics of the target machine. The field containing the primary machine type name (`mc_pmt` in the `target` structure) contains character data, but the field type is defined as `long int`.

```
#include <sys/target.h>

main()
{
     struct target data;

     if (target(MC_GET_TARGET, &data) == -1) {
          perror("target failed");
          exit(1);
     }

     printf("Primary machine type name = %s\n", &data.mc_pmt);
     printf("Number of memory banks = %ld\n", data.mc_bank);
     printf("Number of started processors = %ld\n", data.mc_ncpu);
     printf("Instruction Buffer Size (words) = %ld\n", data.mc_ibsz);
     printf("Main memory size (words) = %ld\n", data.mc_msz);
     printf("Number of clocks for a memory read = %ld\n", data.mc_mspd);
     printf("Clock period in picoseconds = %ld\n", data.mc_clk);
     printf("Number of cluster register sets = %ld\n", data.mc_ncl);
     printf("Memory bank busy time in clocks = %ld\n", data.mc_bbsy);
     printf("Number of clock ticks per second = %ld\n", data.mc_clktck);
     printf("System serial number = %ld\n", data.mc_serial);
     printf("UNICOS release level = %ld\n", data.mc_rls/1000);
}
```

SEE ALSO

`target`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`tcgetpgrp`, `tcsetpgrp` – Gets or sets terminal process group ID of the foreground process group

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp (int fildes);

pid_t tcsetpgrp (int fildes, pid_t pgrp_id);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `tcgetpgrp` system call returns the value of the process group ID of the foreground process group; the `tcsetpgrp` system call sets the foreground process group ID to *pgrp_id*.

The `tcgetpgrp` and `tcsetpgrp` system calls accept the following arguments:

fildes Specifies the controlling terminal of the calling process, and that controlling terminal must be currently associated with the session of the calling process.

pgrp_id Matches a process group ID of a process in the same session as the calling process.

NOTES

The `tcgetpgrp` system call is allowed from a process that is a member of a background process group; however, the information may subsequently be changed by a process that is a member of a foreground process group.

RETURN VALUES

If `tcgetpgrp` completes successfully, it returns the process group ID of the foreground process group associated with the terminal; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

If `tcsetpgrp` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `tcgetpgrp` and `tcsetpgrp` system calls

NAME

_tfork – Creates a multitasking process

SYNOPSIS

```
#include <unistd.h>
int _tfork (void);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The _tfork system call creates a process much as fork(2) does. The main difference between fork(2) and _tfork is that a process created by _tfork shares the same memory area as the parent process. Because the calling process and the created process share the same memory area, the two processes have a sibling-sibling relationship rather than a parent-child relationship. The two processes are said to share a multitasking group. These processes have the following differences from normal processes:

- The process ID (PID) returned when the last process in the multitasking group exits is the *pid* of the first process to exist in the group. The parent *pid* of all processes in the multitasking group is the parent *pid* of the first process in the group. Only the last process in the group can be detected by the wait(2) call. Each process, except the last one to exit, does so without signaling its parent process.
- Whenever a process from a multitasking group is connected to a physical CPU, the process has a cluster. The cluster is loaded when the first process from the group is connected, and it remains loaded as long as any process in the group is connected.

The restart(2) system call and any of the exec(2) family of system calls are not allowed during multitasking. Using them results in the EINVAL error.

RETURN VALUES

If _tfork completes successfully, it returns to each process its own *pid*. If _tfork fails, a value of −1 is returned. Because the two processes share a memory area, a call to _tfork from C does not function as expected, because the stack is not copied; therefore, _tfork is most commonly called from a multitasking library.

FILES

/usr/include/unistd.h	Contains C prototype for the _tfork system call

SEE ALSO

exec(2), fork(2), restart(2), wait(2)

NAME

`thread` – Registers this process as a thread

SYNOPSIS

```
#include <sys/types.h>
#include <sys/thread.h>

int thread (struct thread *buf);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `thread` system call registers this process as a thread and requests special handling by the kernel. It accepts the following argument:

buf Specifies the address of the thread communication area.

The library uses `_tfork`(2) and this system call to implement microtasking. The `thread` structure and context structure are used for fast communication between the library and the kernel. A `thread` structure includes the following members:

```
long   pid;       /* Pid of this process */
long   wakeup;    /* Request by library to wakeup this proc */
long   giveup;    /* Request by kernel to give up cpu */
long   context;   /* Pointer to context save area */
```

The `wakeup` flag may be set by a sibling process in a multitasking group to request that the kernel wake up a sleeping sibling. The `giveup` flag is set by the kernel to request that the thread voluntarily give up the CPU. This is done so that the thread may get to a convenient stopping point and thereby allow the other threads to progress. If the thread does not give up the CPU promptly after `giveup` is set, the kernel will take the CPU.

If the process at any time sets the context pointer to refer to an area outside its address space or shrinks its address space by using `sbreak`(2) so that the `thread` structure is no longer included, the kernel revokes the thread status of the process and sends the `SIGERR` signal to the process.

RETURN VALUES

If `thread` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `thread` system call fails if one of the following error conditions occurs:

Error Code	**Description**
EBUSY	The process is already a thread.
EFAULT	The `thread` structure to which *buf* points is not fully contained in the process address space.
EINVAL	The `pid` value in the `thread` structure is not the correct value for this process.

SEE ALSO

`sbreak`(2), `_tfork`(2)

NAME

`time` – Gets time

SYNOPSIS

```
#include <time.h>
time_t time (time_t *tloc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `time` system call returns the value of time in seconds since 00:00:00 Greenwich mean time (GMT), January 1, 1970. It accepts the following argument:

tloc Points to a second location where the return value is stored.

If *tloc* is 0, `time` returns the time only as the return value. If the *tloc* argument points to an address that is not valid, the actions for `time` are undefined.

NOTES

Under UNICOS, `time` is implemented as a system call, but the `time`(3C) function is also defined to be a part of the ANSI Standard C library. For this reason, this documentation appears both here and in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080.

RETURN VALUES

The `time` system call returns the value of time.

FORTRAN EXTENSIONS

The `time` system call can be called from Fortran as a function:

```
INTEGER TIME, I
I = TIME ( )
```

EXAMPLES

The following example shows how to use the `time` system call to retrieve the current time from the system. It also illustrates how the value returned by `time` is converted to character-string format in two different ways using the `ctime`(3C) and `localtime`(3C) (see `ctime`(3C)) library routines.

```
#include <time.h>
#include <sys/types.h>

main()
{
     static char *daytab[] = {"Sunday", "Monday", "Tuesday",
                   "Wednesday", "Thursday", "Friday", "Saturday"};
     time_t timval;
     struct tm *tmptr;

     time(&timval);
     printf("The time in seconds since Jan 1, 1970 is %ld\n", timval);

     printf("The date and time are %s", ctime(&timval));

     tmptr = localtime(&timval);
     printf("The reformatted date and time are %s %2d/%2d/%2d %.2d:%.2d\n",
            daytab[tmptr->tm_wday], tmptr->tm_mon + 1, tmptr->tm_mday,
            tmptr->tm_year, tmptr->tm_hour, tmptr->tm_min);
}
```

SEE ALSO

`stime`(2)

`ctime`(3C), `time`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

times – Gets process and child process times

SYNOPSIS

```
#include <sys/times.h>
clock_t times (struct tms *buffer);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The times system call returns time-accounting information to the process. It accepts the following argument:

buffer Points to the tms structure.

A tms structure includes the following members:

```
clock_t   tms_utime;     CPU time used during the execution
                         of instructions in the user space
                         of the calling process
clock_t   tms_stime;     CPU time used by the system
                         on behalf of the calling process
clock_t   tms_cutime;    Sum of the "tms_utime"s and "tms_cutime"s
                         of the child processes
clock_t   tms_cstime;    Sum of the "tms_stime"s and "tms_cstime"s
                         of the child processes
```

This information comes from the calling process and each of its terminated child processes for which it has executed a wait(2). All times are given in system hardware clock ticks; there are CLK_TCK system hardware clock ticks per second. The CLK_TCK macro is defined in the time.h file.

RETURN VALUES

If times completes successfully, it returns the elapsed real time, in system hardware clock ticks, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of times to another. If times fails, a –1 is returned, and errno is set to indicate the error.

ERRORS

The `times` system call fails if the following error condition occurs:

Error Code	Description
`EFAULT`	The *buffer* argument points to an illegal address.

FORTRAN EXTENSIONS

The `times` system call can be called from Fortran as a function:

```
INTEGER buffer(n), TIMES, I
I = TIMES (buffer)
```

EXAMPLES

This example shows how to use the `times` system call to gather CPU usage information to time a particular section of user code:

```
#include <sys/times.h>
#include <time.h>

main()
{
     struct tms before, after;
     clock_t utime, stime, startime, endtime;
     startime = times(&before);

     /* The section of code to be timed resides here. */

     endtime = times(&after);

     utime = after.tms_utime - before.tms_utime;
     stime = after.tms_stime - before.tms_stime;

     printf("\nCPU time used in user space = %f sec or %ld clock ticks\n",
               (float)utime/(float)CLK_TCK, utime);
     printf("CPU time used by the system = %f sec or %ld clock ticks\n",
               (float)stime/(float)CLK_TCK, stime);
     printf("Wall clock time used by process = %f sec ",
               (float)(endtime - startime)/(float)CLK_TCK);
     printf("or %ld clock ticks\n", endtime - startime);
}
```

SEE ALSO

`exec`(2), `fork`(2), `time`(2), `wait`(2)

NAME

trunc – Truncates a file

SYNOPSIS

```
#include <unistd.h>
long trunc (int fildes);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The trunc system call sets the size of the file indicated by *fildes* to the current file pointer. The process must have write permission to the file. The trunc system call accepts the following argument:

fildes Indicates the size of the file.

NOTES

In addition to changing the size of a file, the trunc system call releases file storage beyond the truncated size, including any storage preallocated to the file through the ialloc(2) system call.

A process is granted write permission to the file only if the active security label of the process is equal the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_MAC_WRITE	The process is granted write permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted write permission to the file.

RETURN VALUES

If trunc completes successfully, the new file size is returned; otherwise, a value of –1 is returned, and errno is set to indicate the error.

ERRORS

The trunc system call fails if one of the following error conditions occurs:

Error Code	**Description**
EAGAIN	Mandatory file and record locking is set (see chmod(2)), outstanding record locks exist on the file (see fcntl(2)), and O_NDELAY was set in the file flag word.
EBADF	The *fildes* argument is not a valid file descriptor open for writing.

EBADF	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
EDEADLK	A deadlock situation would have occurred waiting for a blocking record lock to be removed.
EINTR	Mandatory file and record locking is set (see chmod(2)), outstanding record locks exist on the file (see fcntl(2)), and O_NDELAY is not set in the file flag word.
EINVAL	The pointer for *fildes* is beyond the end-of-file.
ENOLCK	The system record lock table was full; therefore, it was not possible to wait for a blocking record lock to be removed.

FORTRAN EXTENSIONS

The trunc system call can be called from Fortran as a function:

```
INTEGER fildes, TRUNC, I
I = TRUNC (fildes)
```

EXAMPLES

This example shows how to use the trunc system call to truncate the last half of a file's contents. In this case, the request truncates file test_data so that the file is one-half of its original size.

```
#include <fcntl.h>
#include <unistd.h>

main()
{
     int fd;
     long size;

     if ((fd = open("test_data", O_RDWR)) == -1) {
          perror("open failed");
          exit(1);
     }

     size = lseek(fd, 0L, 2);    /* determine size of the file */

     lseek(fd, size/2, 0);       /* seek to middle of the file */

     if (trunc(fd) == -1) {      /* truncate last half of the file */
          perror("trunc failed");
          exit(1);
     }

     close(fd);
}
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `trunc` system call

SEE ALSO

`chmod`(2), `fcntl`(2), `ialloc`(2), `lseek`(2)

NAME

`ulimit` – Gets and sets user limits

SYNOPSIS

```
#include <ulimit.h>
long int ulimit (int cmd, ...);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `ulimit` system call controls process limits. It accepts the following argument:

cmd Specifies one of the values, defined in the `ulimit.h` file. These values are as follows:

`UL_GETFSIZE` Gets the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

`UL_SETFSIZE` Sets the file size limit of the process to the value of the second argument, taken as a `long int`. Any process can decrease this limit, but only a process with an effective user ID of the super user can increase the limit. The new file size limit is returned.

`UL_GMEMLIM` Gets the maximum break value in bytes. On Cray PVP systems, this value is an integer number of bytes; on Cray MPP systems, it is the actual byte address of the break value. To use this value as an argument to the `brk`(2) system call, see example 2 in the EXAMPLES section.

Only an appropriately privileged process can increase a file size limit.

NOTES

The minimum allocation unit, both on disk and in memory, for all Cray Research systems is 4096 bytes. When `ulimit` is called to set the process limit, the limit is rounded to the next 4096-byte boundary. (For example, if `ulimit` is called to set the limit at 5120 bytes, it is actually set to 8192 bytes.)

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_RESOURCE`	The process is allowed to increase a file size limit.

If the PRIV_SU configuration option is enabled, the super user or a process with the PERMBITS_RESLIM permbit is allowed to increase a file size limit.

RETURN VALUES

If ulimit completes successfully, a nonnegative value is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

If the following error condition occurs when the value of *cmd* is UL_SETFSIZE, the ulimit system call fails and the process limit remains unchanged.

Error Code	Description
EINVAL	An illegal argument was passed to the system call.
EPERM	A process without appropriate privilege tried to increase the file size limit.

FORTRAN EXTENSIONS

The ulimit system call can be called from Fortran as a function:

```
INTEGER cmd, newlimit, ULIMIT, I
I = ULIMIT (cmd, newlimit)
```

EXAMPLES

The following examples illustrate use of the ulimit system call to get and set user limits.

Example 1: This ulimit request returns the file size limit for the current process. Because the file size limit value is in 512-byte units, it is converted to the more familiar unit of 512 words.

```
#include <ulimit.h>

main()
{
     long fslim;

     fslim = ulimit(UL_GETFSIZE);
     printf("File size limit = %ld (512-byte) blocks\n", fslim);
     printf("                = %ld (512-word) blocks\n", fslim/8);
     }
```

Example 2: This `ulimit` request returns the maximum break value for this process; then the `brk` system call attempts to increase the process size to that limit.

```
#include <ulimit.h>

main()
{
     if  ((brk(((char *)0) + ulimit(UL_GMEMLIM))) == -1) {
            perror("brk failed");
            exit(1);
     }
}
```

FILES

`/usr/include/ulimit.h`	Contains C prototype for the `ulimit` system call; also contains the `UL_GETFSIZE`, `UL_SETFSIZE`, and `UL_GMEMLIM` symbols.

SEE ALSO

`brk`(2), `limit`(2), `write`(2)

NAME

umask – Sets and gets file creation mask

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask (mode_t cmask);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `umask` system call sets the file creation mode mask of the process to *cmask* and returns the previous value of the mask.

The `umask` system call accepts the following argument:

cmask Specifies the new value of the file creation mode mask. Only the low-order 9 bits of *cmask* and the file creation mode mask are used.

RETURN VALUES

The previous value of the file mode creation mask is returned.

FORTRAN EXTENSIONS

The `umask` system call can be called from Fortran as a function:

```
INTEGER cmask, UMASK, I
I = UMASK (cmask)
```

EXAMPLES

This example shows how to use the `umask` system call to change a process's file creation mask. The following `umask` request changes the file creation mask of the current process to 077, and the previous file creation mask is displayed.

After the file creation mask is altered, an `open` request creates a file with permissions of 0755. Because the file creation mask is now 077, the permissions set for the new file are 0700.

```
main()
{
     printf("The previous file creation mask was %o\n", umask(077));

     if ((fd = open("datafile", O_CREAT | O_WRONLY, 0755)) == -1) {
          perror("open failed");
          exit(1);
     }
}
```

SEE ALSO

chmod(2), creat(2), mknod(2), open(2)

mkdir(1), ksh(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`umount` – Unmounts a file system

SYNOPSIS

`int umount (char *`*file*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `umount` system call accepts the following argument:

file Points to a path name.

The `umount` system call requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted; *file* is a pointer to a path name. After unmounting the file system, the directory on which the file system was mounted reverts to its ordinary interpretation.

Only an appropriately privileged process can use this system call.

NOTES

Unmounting the root device causes the kernel to reread all in-core (in memory) information from that device.

A process is granted search permission to a component of the path prefix only if the active security label of the process is greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	Description
`PRIV_ADMIN`	The process is allowed to use this system call.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call and is granted search permission to every component of the path prefix.

RETURN VALUES

If `umount` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `umount` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Search permission is denied for a component of the path prefix.
`EBUSY`	A file on *file* is busy.
`EFAULT`	The *file* argument points outside the allocated process address space.
`EINVAL`	The *file* argument is not mounted.
`ENAMETOOLONG`	The lenth of the *file* argument exceeds `PATH_MAX`, or a path name component exceeds `NAME_MAX` while `POSIX_NO_TRUNC` is in effect.
`ENOENT`	The specified file does not exist or the *file* argument points to an empty string.
`ENOTDIR`	A component of the path prefix is not a directory.
`EPERM`	The process does not have appropriate privilege to use this system call.

SEE ALSO

`mount`(2)

NAME

`uname` – Gets name of current operating system

SYNOPSIS

```
#include <sys/utsname.h>
int uname (struct utsname *name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `uname` system call stores information identifying the current operating system. It accepts the following argument:

name Points to the structure to receive the information. Each member of the structure receives a null-terminated character string.

A `utsname` structure includes the following members:

```
char   sysname[9];   /* Current operating system name */
char   nodename[9];  /* Name by which the system is known
                        on a communications network */
char   release[9];   /* Release of the operating system */
char   version[9];   /* Release version of the operating system */
char   machine[12];   /* Standard name identifying the hardware
                        on which the operating system is running */
```

RETURN VALUES

If `uname` completes successfully, a nonnegative value is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `uname` system call fails if the following error condition occurs:

Error Code	Description
`EFAULT`	The *name* argument points to an address that is not valid.

FORTRAN EXTENSIONS

See UNAME(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165 (for all systems except Cray MPP systems and CRAY T90 series systems). Also see the PXFUNAME(3F) subroutine.

EXAMPLES

This example shows how to use the uname system call to retrieve the name of the operating system as well as the release and version of the operating system:

```
#include <sys/utsname.h>

main()
{
     struct utsname opname;

     if (uname(&opname) == -1) {
          perror("uname failed");
          exit(1);
     }
     else {
          printf("The current operating system is %s\n", opname.sysname);
          printf("    Release %s\n", opname.release);
          printf("    Version %s\n", opname.version);
     }
}
```

SEE ALSO

uname(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

PXFUNAME(3F), UNAME(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`unlink`, `unlink2` – Removes directory entry

SYNOPSIS

All Cray Research systems:

```
#include <unistd.h>

int unlink (const char *path);
```

Cray PVP systems:

```
#include <unistd.h>
int unlink2 (const char *path);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4 (applies only to `unlink`)

DESCRIPTION

The `unlink` system call removes the directory entry specified by the path name to which the *path* argument points. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

The `unlink2` system call, which is a Cray Research extension, functions like the `unlink` system call except for the values returned.

The `unlink` and `unlink2` system calls accept the following argument:

path Points to the path name of the directory entry to be removed.

The values returned by the `unlink2` system call differ from those returned by `unlink`. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. In this case, `unlink2` returns a positive value that represents the number of blocks of space returned to the file system free space pool.

If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file are closed. In this case, `unlink2` returns a 0 if the operation is allowed, and the actual file space is returned later.

NOTES

The `unlink` system call does not remove a directory from the file system, it simply unlinks the reference from the specified directory. Use of `unlink` on directories by privileged users can cause file system errors (unlinked inodes), which can be fixed by using the `fsck`(8) command. A privileged user should use `rmdir`(2) to remove a directory from the file system.

A process is granted write permission to the directory containing the link only if the active security label of the process is equal to the security label of the directory.

A process is granted search permission to a component of the path prefix only if the active security label of the process is greater than or equal to the security label of the component.

The process must be granted write permission to the file via the active security label. That is, the security label of the process must equal the security label of the specified file.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_ADMIN`	The process is allowed to unlink a directory.
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the file's parent directory via the permission bits and access control list.
`PRIV_FOWNER`	The process is allowed to specify a directory that has the "sticky" mode bit set and that the process does not own.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the specified file and its parent directory via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to unlink a directory. The super user is allowed to specify a directory that has the "sticky" mode bit set and that it does not own. The super user is granted search permission to every directory component of the path prefix. The super user is granted write permission to the file and its parent directory. If the `PRIV_SU` configuration option is enabled, the super user is granted write permission to the file via the security label.

RETURN VALUES

If `unlink` completes successfully, a value of 0 is returned.

If `unlink2` completes successfully and the file space has already been returned to the file system free space pool, a positive value that represents the number of blocks of space returned is returned. If the actual return of the file space to the file system free pool has been postponed because some other process still references the file, then a value of 0 is returned.

If unlink or unlink2 fail to complete successfully, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The unlink or unlink2 system call fails and the specified file remains linked if one of the following error conditions occurs:

Error Code	Description
EACCES	Search permission is denied for a component of the path prefix.
EACCES	Write permission is denied on the directory containing the link to be removed.
EACCES	The active security label of the process does not equal the specified security label of the file.
EBUSY	The entry to be unlinked is the mount point for a mounted file system.
EFAULT	The *path* argument points outside the allocated process address space.
ENAMETOOLONG	The *path* argument is longer than PATH_MAX characters.
ENOENT	The specified file does not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The specified file is a directory, and the process does not have appropriate privilege.
EROFS	The directory entry to be unlinked is part of a read-only file system.

FORTRAN EXTENSIONS

The unlink system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER UNLINK, I
I = UNLINK (path)
```

Alternatively, unlink can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
CALL UNLINK (path)
```

The Fortran program must not specify both the subroutine call and the function reference to unlink() from the same procedure. *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The PXFUNLINK(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example shows how to use the `unlink` system call to implement a scratch file for use in the program. A unique name for the scratch file is derived by calling the `tmpnam` subroutine. The `unlink` request unlinks the scratch file immediately after it is opened. At this point, the file has no links and is called a *zero-linked file*.

The scratch file (possessing no links) is not removed because the program still has it open for access. The scratch file remains in existence until the program closes it, terminates without closing it, or abnormally terminates, or until the UNICOS system dies.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
     int fd;
     char *scratch;                /* path name to a scratch file */

     scratch = tmpnam((char *) 0); /* create unique temp file name */

     /* Create a file; open it for read & write. */

     if( (fd = open(scratch, O_RDWR | O_CREAT | O_EXCL, 0600)) == -1) {
          perror("open failed");
          exit(1);
     }

     /* Now remove links, but don’t close it. */

     if (unlink(scratch) == -1) {
          perror("unlink failed");
          exit(1);
     }

     /* Program writes and reads the file here. */

     close(fd);                     /* also removes file, since # links = 0 */
}
```

FILES

`/usr/include/unistd.h`	Contains C prototype for the `unlink` and `unlink2` system calls

SEE ALSO

`close`(2), `link`(2), `open`(2), `rmdir`(2)

`rm`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`PXFUNLINK`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

`fsck`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`upanic` – Stops the system from a user process

SYNOPSIS

```
#include <sys/panic.h>
int upanic (int cmd);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `upanic` system call, which is referred to as the user panic, allows the system to be stopped from a user process. This feature is useful with problems, such as bad data on an I/O read, that cannot be detected at the system level or for problems that occur only with a specific user code or activity, such as user data corruption. The `upanic` system call accepts the following argument:

cmd Specifies an entry. It can be one of the following:

`PA_SET` Sets the user panic flag; requires appropriate privilege.

`PA_RELAX` Clears the user panic flag; requires appropriate privilege.

`PA_PANIC` Stops the system if the user panic flag has been set; can be called by any process.

When `PA_PANIC` is sent but the user panic flag is not set, the call is inoperative; thus, it can be embedded in code with no side effect other than the overhead of the system call path.

Only an appropriately privileged process can set or clear the user panic flag.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	Description
`PRIV_ADMIN`	The process is allowed to set or clear the user panic flag.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to set or clear the user panic flag.

RETURN VALUES

When `upanic` completes successfully, a value of 0 is returned; otherwise a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `upanic` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EINVAL`	An argument is not valid. The command is not one of the listed values.
`EPERM`	The process does not have appropriate privilege to set or clear the user panic flag.

SEE ALSO

`panic`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`ustat` – Gets file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev_t dev, struct ustat *buf);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `ustat` system call returns information about a mounted file system. It accepts the following arguments:

dev Specifies a device number that identifies a device containing a mounted file system.

buf Points to a `ustat` structure.

The `ustat` structure includes the following members:

```
daddr_t   f_tfree;      /* Total free blocks */
ino_t     f_tinode;     /* Number of free inodes */
char      f_fname[6];   /* Name of the mounted file system */
char      f_fpack[6];   /* Name of the file system pack */
```

NOTES

The `statfs`(2) system call obsoletes some purposes of `ustat`, but `ustat` remains useful for determining whether a given device is mounted.

RETURN VALUES

If `ustat` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `ustat` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EFAULT`	The *buf* argument points outside the allocated process address space.
`EINVAL`	The *dev* argument is not the device number of a device containing a mounted file system.

FORTRAN EXTENSIONS

The `ustat` system call can be called from Fortran as a function:

```
INTEGER dev, buf(m), USTAT, I
I = USTAT (dev, buf)
```

SEE ALSO

`stat`(2), `statfs`(2)

`fs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`utime` – Sets file access and modification times

SYNOPSIS

```
#include <sys/types.h>
#include <utime.h>

int utime (const char *path, const struct utimbuf *times);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `utime` system call sets the access and modification times of a specified file. It accepts the following arguments:

path Points to a file path name.

times Specifies source of the access and modification times.

If *times* is null, the access and modification times of the file are set to the current time. A process must be the file owner or have write permission to use `utime` in this manner.

If *times* is not null, it is interpreted as a pointer to a `utimbuf` structure, and the access and modification times are set to the values contained in the designated structure. Only the file owner can use `utime` this way.

The `utimbuf` structure follows:

```
struct utimbuf   {
    time_t   actime;    /* Access time */
    time_t   modtime;   /* Modification time */
};
```

Times are measured in seconds since 00:00:00 Greenwich mean time (GMT), January 1, 1970.

The `utime` function also causes the time of the last file status change (`st_ctime`) to be updated (see `stat`(2)).

NOTES

A process is granted write permission to the file only if the active security label of the process is equal to the security label of the file.

A process is granted search permission to a component of the path prefix only if the active security label of the process is greater than or equal to the security label of the component.

A process with the effective privileges shown is granted the following abilities:

Privilege	**Description**
`PRIV_DAC_OVERRIDE`	The process is granted search permission to every component of the path prefix via the permission bits and access control list.
`PRIV_DAC_OVERRIDE`	The process is granted write permission to the file's parent directory via the permission bits and access control list.
`PRIV_FOWNER`	The process is considered the file owner.
`PRIV_MAC_READ`	The process is granted search permission to every component of the path prefix via the security label.
`PRIV_MAC_WRITE`	The process is granted write permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is considered the file owner, is granted search permission to every component of the path prefix, and is granted write permission to the file.

RETURN VALUES

If `utime` completes successfully, a value of 0 is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `utime` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EACCES`	Search permission is denied by a component of the path prefix.
`EACCES`	The process is not the file owner, *times* is null, write permission is denied, and the process does not have appropriate privilege.
`EFAULT`	The *path* argument points outside the allocated process address space.
`EFAULT`	The *times* argument is not null and points outside the allocated process address space.
`EMANDV`	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
`ENOENT`	The specified file does not exist.
`ENOTDIR`	A component of the path prefix is not a directory.

`EPERM` The process is not the file owner, *times* is not null, and the process does not have appropriate privilege.

`EROFS` The file system containing the file is mounted as read only.

FORTRAN EXTENSIONS

The `utime` system call can be called from Fortran as a function (on all systems except Cray MPP systems and CRAY T90 series systems):

```
CHARACTER*n path
INTEGER times, UTIME, I
I = UTIME (path, times)
```

Alternatively, `utime` can be called from Fortran as a subroutine (on all systems except Cray MPP systems and CRAY T90 series systems). In this case, the return value of the system call is unavailable.

```
CHARACTER*n path
INTEGER times
I = UTIME (path, times)
```

The Fortran program must not specify both the subroutine call and the function reference to `utime` from the same procedure. *path* may also be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte. The `PXFUTIME`(3F) subroutine provides similar functionality and is available on all Cray Research systems.

EXAMPLES

This example shows how to use the `utime` system call to modify the last accessed and last modified time-stamps in a file's inode.

The program first displays the current time stamps saved in the file's inode. Then, the `utime` request modifies the two time stamps, and they are displayed again.

```
#include <sys/types.h>
#include <utime.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>

main()
{
     static char file[] = {"datafile"};
     struct stat buf;

     if (stat(file, &buf) == -1) {
          perror("stat failed");
          exit(1);
     }

     printf("Before utime(), %s was last accessed on %s",
                                               file, ctime(&buf.st_atime));
     printf("Before utime(), %s was last modified on %s",
                                               file, ctime(&buf.st_mtime));

     if (utime(file, ((struct utimbuf *) 0)) == -1) {  /* set timestamps to */
          perror("utime failed");                      /* current time      */
          exit(1);
     }

     if (stat(file, &buf) == -1) {
          perror("stat failed");
          exit(1);
     }

     printf("\nAfter utime(), %s was last accessed on %s",
                                               file, ctime(&buf.st_atime));
     printf("After utime(), %s was last modified on %s",
                                               file, ctime(&buf.st_mtime));
}
```

SEE ALSO

stat(2)

PXFUTIME(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

`vfork` – Creates a new process in a memory efficient way

SYNOPSIS

```
#include <unistd.h>
int vfork (void);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `vfork` system call can be used to create new processes without fully copying the address space of the old process. It is useful when the purpose of `fork`(2) would have been to create a new system context for an `execv`(2). The `vfork` system call differs from `fork`(2) in that the child borrows the parent's memory and thread of control until a call to `execve`(2) or an exit (either by a call to `exit`(2) or an abnormal exit). The parent process is suspended while the child is using its resources.

The `vfork` system call returns 0 in the child's context and (later) the process ID of the child in the parent's context.

The `vfork` system call can normally be used just like `fork`. It does not work, however, to return while running in the child's context from the procedure that called `vfork` since the eventual return from `vfork` would then return to a no longer existent stack frame. Be careful to call `_exit`(2) rather than `exit`(2) if you cannot call `execve`(2), because `exit`(2) will flush and close standard I/O channels, and mess up the parent process's standard I/O data structures. (Even when using `fork`(2), it is wrong to call `exit`(2) because buffered data would then be flushed twice.)

RETURN VALUES

If `vfork` completes successfully, it returns a value of 0 to the child process and returns the process ID of the child process to the parent process; otherwise, a value of –1 is returned to the parent process, no child process is created, and `errno` is set to indicate the error.

ERRORS

The `fork` system call fails and no child process is created if one of the following error conditions occurs:

Error Code	Description
`EAGAIN`	The system-imposed limit on the total number of processes under execution in the whole system (`NPROC`) is exceeded.
`EAGAIN`	The system-imposed limit on the total number of processes under execution by one user (`CHILD_MAX`) is exceeded.
`ENOMEM`	Not enough main memory or swap space exists.

BUGS

Because UNICOS `signal`(2) and `sigctl`(2) signal registration is implemented with a library-level signal vector, any changes in signal registration by the child will be reflected in the parent process. This behavior differs from other UNIX systems supporting the `vfork` system call. Other changes to signal disposition (for example, `SIG_IGN` or `SIG_DFL`) will behave the same as with the `fork`(2) system call.

EXAMPLES

The following examples illustrate different uses of the `vfork` system call.

Example 1: The `vfork` request generates a new process (referred to as the child process). The child process returns from `vfork` and executes in the same process space as the parent process. The parent process does not return from the `vfork` request until the child process has executed some form of `exec`(2) request or an exit. At the time that the child process issues an `exec`(2) request, enough memory is generated for the new (child) process to execute the specified program; then the parent process returns from `vfork` and continues execution. The return value from `vfork` indicates whether execution is in the parent or child process.

```
int res;

if ((res = vfork()) == -1) {
    perror("vfork failed");
    exit(1);
}

if (res == 0) {
     /* Code here is executed in the child process until an exec or
        _exit request is made.  Parent does not return from vfork
        until child process issues one of these requests.  Since child
        process has access to parent's data fields and signal
        dispositions here until an exec or _exit request, it should
        not modify those on which the parent depends.  Child process
        must refrain from returning (e.g., falling out of the process)
        since that will cause the parent process' stack frame to be
        removed.  Parent process expects presence of the stack frame. */

}

else {
     /* Code here is executed in the parent process after the child
        process issues an exec or _exit request. */
}
```

Example 2: This example illustrates a typical usage of the `vfork` request. When a parent process generates a child process so that a different program can execute in the child process, the `vfork` request is the most efficient way to handle the task.

Typically, when the child process returns from `vfork`, it immediately performs an `exec`(2) request (in this case `execl`(2)) to generate a new process space and to load the specified program for execution into the child process. With `vfork`, the process space for the parent is not duplicated in the child process. The parent and child processes then execute different programs in parallel.

```
int res;

if ((res = vfork()) == -1) {
     perror("vfork failed");
     exit(1);
}

if (res == 0) {             /* In child process? */
    execl("childprog", "childprog", "arg1", "arg2", 0);
    perror("exec for childprog failed");
    _exit(1);
}

/* Parent process continues execution here after successful execl
   request in the child process. */
```

FILES

`/usr/include/unistd.h` Contains C prototype for the `vfork` system call

SEE ALSO

`exec`(2), `fork`(2), `sigctl`(2), `signal`(2), `wait`(2)

NAME

`wait`, `waitpid` – Waits for a child process to stop or terminate

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *stat_loc);

pid_t waitpid (pid_t pid, int *stat_loc, int options);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `wait` system call suspends the calling process until one of the child processes terminates or until a child process that is being traced stops because it has hit a breakpoint. If a signal is received, `wait` returns prematurely. If a child process has stopped or terminated before the call on `wait`, return is immediate.

The `wait` and `waitpid` system calls accept the following arguments:

stat_loc Returns the status of a terminated child process.

pid Specifies the child process that will have its status returned. -1 indicates that the status of any available terminated process should be returned.

options Sets optional flag for the `waitpid` system call. The *options* argument is constructed from the bitwise inclusive OR of 0 or more of the following flags, defined in header file `sys/wait.h`:

`WNOHANG` Indicates that `waitpid` returns immediately and does not suspend execution of the calling process if status is not available for one of the child processes specified by *pid*.

`WUNTRACED` Provides the following status if job control is supported. Reports to the requesting process the status of any child process specified by *pid* that has stopped and whose status has not yet been reported since it stopped.

`WMTWAIT` Waits for the children of any member of the multitasking group. In UNICOS 9.0 this is the default behavior for both `wait` and `waitpid`. The flag is still provided for source compatibility. To get the previous behavior, see the description of the `WLWPWAIT` flag.

`WLWPWAIT` Waits only for the immediate children of the calling light-weight process (LWP). This flag is not recommended for general use.

If 0, the caller will suspend until a child process stops or terminates.

If the *stat_loc* argument is not 0, 16 bits of status information are stored in the low-order 16 bits of the location to which *stat_loc* points. This status differentiates between stopped and terminated child processes. If the child process has terminated, The status identifies the cause of termination and passes useful information to the parent process. This is accomplished in the following manner:

- If the child process has stopped, the high-order 8 bits of status contain the number of the signal that caused the process to stop and the low-order 8 bits are set equal to 0177.
- If the child process has terminated because of an `exit`(2) call, the low-order 8 bits of status are 0 and the high-order 8 bits contain the low-order 8 bits of the argument that the child process passed to `exit`(2).
- If the child process has terminated because of a signal, the high-order 8 bits of status are 0 and the low-order 8 bits contain the number of the signal that caused the termination. If the low-order seventh bit (that is, bit 0200) is set, a core image will also have been produced; see `signal`(2).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means that the initialization process inherits the child processes.

The `waitpid` system call behaves identically to `wait` if the *pid* argument has a value of –1 and the *options* argument has a value of 0; otherwise, the values of *pid* and *options* modify its behavior.

The *pid* argument specifies a set of child processes for which status is requested. The `waitpid` system call returns only the status of a child process from this set.

- If *pid* is equal to –1, status is requested for any child process; `waitpid` is then equivalent to `wait`.
- If *pid* is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If *pid* is equal to 0, status is requested for any child process with a process group ID that is equal to that of the calling process.
- If *pid* is less than –1, status is requested for any child process with a process group ID that is equal to the absolute value of *pid*. The *options* argument is constructed from the bitwise inclusive OR of 0 or more of the following flags, defined in header file `sys/wait.h`:

If `wait` and `waitpid` return because the status of a child process is available, these system calls return a value equal to the process ID of the child process. In this case, if the value of the *stat_loc* argument is not `NULL`, information is stored in the location to which *stat_loc* points. If, and only if, the status returned is from a terminated child process that returned a value of 0 from `main`( ) or passed a value of 0 as the *status* argument to `_exit`(2) or `exit`(2), the value stored at the location to which *stat_loc* points is 0.

Regardless of its value, this information is interpreted using macros. These macros are defined in the `sys/wait.h` file and evaluate to integral expressions. The *stat_val* argument is the integer value to which *stat_loc* points.

`WIFEXITED` (*stat_val*) Returns a nonzero value if the child process terminated normally.

`WEXITSTATUS` (*stat_val*)	Determines the low-order 8 bits of the argument that the child process passed to `_exit`(2) or `exit`(2), or the value the child process returned from `main`( ). Use only if `WIFEXITED` returns a nonzero value.
`WIFSIGNALED` (*stat_val*)	Returns a nonzero value if the child process terminated due to the receipt of a signal that it did not catch (see the `signal.h` file).
`WTERMSIG` (*stat_val*)	Determines the number of the signal that caused the termination of the child process. Use only if `WIFSIGNALED` returns a nonzero value.
`WIFSTOPPED` (*stat_val*)	Returns a nonzero value if the child process is stopped due to a signal.
`WSTOPSIG` (*stat_val*)	Determines the number of the signal that caused the child process to stop. Use only if `WIFSTOPPED` returns a nonzero value.

If the information in the location to which *stat_loc* points is stored there by a call to `waitpid` that specified the `WUNTRACED` flag, exactly one of the `WIFEXITED`, `WIFSIGNALED`, and `WIFSTOPPED` macros evaluates to a nonzero value. If the information stored at the location to which *stat_loc* points is stored there by a call to `waitpid` that did not specify the flag or a call to `wait`, exactly one of the `WIFEXITED` and `WIFSIGNALED` macros evaluates to a nonzero value.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes (now orphaned) are assigned a new parent process ID. The parent process of orphaned child processes is the init process (*pid* = 1).

NOTES

In UNICOS 9.0, the default behavior of both `wait` and `waitpid` acts as though the `WMTWAIT` flag was set. The `WLWPWAIT` flag provides the previous default behavior. However, it is not expected that this will be useful because using `waitpid` with a specified process ID should provide the necessary control for child process management.

The idea of a parent process has changed in UNICOS 9.0. Previously, the parent was the entity (previously termed a *process*, now a *light-weight process*) that created the child by using the `fork`(2) system call. Now, the parent process is the entire multitasking group in which the former parent process was a member. This change is part of the more general change that moves from a multitasking model that supports multiple processes in a multitasking group to a model that supports a single process. This change is described more fully in the `getpid`(2) man page.

RETURN VALUES

If the child process stopped or terminated after the parent process's call to `wait`, the system call returns the child process ID. If `wait` is interrupted by a signal other than the death-of-a-child-process signal (`SIGCLD`) or if the calling process has no existing zombie-producing child processes (see the following paragraph), a value of −1 is returned, and `errno` is set to indicate the error.

A zombie-producing child process results when the death-of-a-child-process signal `SIGCLD` is set to anything other than to be ignored. If `SIGCLD` is set to be ignored, a call to `wait` returns −1, and an `errno` of `ECHILD`.

If `wait` or `waitpid` returns because the status of a child process is available, the call returns a value equal to the process ID of the child process for which status is reported. If `wait` or `waitpid` returns due to the delivery of a signal to the calling process, a value of −1 is returned and `errno` is set to `EINTR`. If the `waitpid` system call is invoked with `WNOHANG` set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `wait` system call fails and its actions are undefined if the *stat_loc* argument points to an illegal address. The call fails and returns immediately if one of the following error conditions occurs:

Error Code	**Description**
`ECHILD`	The calling process has no existing unwaited-for child processes.
`EINTR`	Receipt of a signal other than the death-of-a-child-process signal.

The `waitpid` system call returns −1, and `errno` is set to indicate the error if one of the following error conditions occurs:

Error Code	**Description**
`ECHILD`	The process or process group specified by *pid* does not exist or is not a child of the calling process.
`EINTR`	The call was interrupted by a signal. The value of the location to which *stat_loc* points is undefined.
`EINVAL`	The value of the *options* argument is not valid.

FORTRAN EXTENSIONS

The `wait` system call may be called from Fortran as a function:

```
INTEGER statloc, WAIT, I
I = WAIT (statloc)
```

EXAMPLES

The following examples illustrate use of the `wait` and `waitpid` system calls. Both examples show a parent process waiting for its child process to complete.

Example 1: In this program, the `wait` request in the parent process waits for its child process to complete.

The program first creates a child process and allows the child process to perform some other work. Executing in parallel with the child process, the parent displays the process identification number (PID) of the forked child process and then waits for its completion. Once the child process completes, the parent uses a macro (that is, `WIFEXITED` or `WIFSIGNALED`) to determine the cause of the child's termination.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

main()
{
     int res, cid_ret, cid_stat;

     if ((res = fork()) == -1) {
          perror("fork failed");
          exit(1);
     }

     if (res == 0) {          /* child process */
          /* child process performs its work here */
     } else {                   /* parent process */
          printf("Child process has pid = %d\n", res);
          cid_ret = wait(&cid_stat);        /* waits for child to complete */
          if (WIFEXITED(cid_stat)) {        /* if child terminated normally */
               printf("Child process %d terminated normally with ", cid_ret);
               printf("exit status = %d.\n", WEXITSTATUS(cid_stat));
          } else {
               if (WIFSIGNALED(cid_stat)) { /* if child terminated (signal) */
                    printf("Child process %d terminated due to ", cid_ret);
                    printf("signal no. -> %d.\n", WTERMSIG(cid_stat));
               }
          }
     }
}
```

Example 2: In this program, the `waitpid` request in the parent process waits for its child process to complete.

The program first creates a child process and allows the child process to perform some other work. Executing in parallel with the child process, the parent displays the PID of the forked child process and then waits for its completion. Once the child process completes, the parent uses a macro (that is, `WIFEXITED` or `WIFSIGNALED`) to determine the cause of the child's termination.

```
#include <unistd.h>
#include <sys/wait.h>

main()
{
     int res, cid_ret, cid_stat;

     if ((res = fork()) == -1) {
          perror("fork failed");
          exit(1);
     }

     if (res == 0) {          /* child process */
          /* child process performs its work here */
     } else {                   /* parent process */
          printf("Child process has pid = %d\n", res);
          cid_ret = waitpid(res, &cid_stat, 0);/* waits for child to complete */
          if (WIFEXITED(cid_stat)) {        /* if child terminated normally */
               printf("Child process %d terminated normally with ", cid_ret);
               printf("exit status = %d.\n", WEXITSTATUS(cid_stat));
          } else {
               if (WIFSIGNALED(cid_stat)) { /* if child terminated (signal) */
                    printf("Child process %d terminated due to ", cid_ret);
                    printf("signal no. -> %d.\n", WTERMSIG(cid_stat));
               }
          }
     }
}
```

SEE ALSO

`exec`(2), `exit`(2), `fork`(2), `getpid`(2), `intro`(2), `pause`(2), `signal`(2)

NAME

`waitjob` – Gets information about a terminated child job

SYNOPSIS

```
#include <sys/types.h>
#include <sys/jtab.h>

int waitjob (struct jtab *jtab);
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `waitjob` system call obtains information about a terminated child job (that is, a child job in which all of the processes have exited) that has been configured to send a signal to its parent on termination. The system call is named `waitjob` because, like the `wait`(2) system call, it returns information about only an object that is considered to be a child of the calling process. Unlike `wait`, however, and despite its name, `waitjob` never blocks the caller's execution.

The `waitjob` system call accepts the following argument:

jtab Returns the *jtab* entry for the terminated job.

If the *jtab* argument is not 0, the `jtab` structure containing statistics for the terminated job is returned at that address; otherwise, no `jtab` structure is returned.

If the parent process of any job exits, the parent process ID of each remaining child job is set to 0, and the jobs exit silently from the system on termination.

The `waitjob` system call obtains information only for a terminated job that was configured to send a signal to its parent on termination. The `setjob`(2) system call makes it possible to create jobs that exit from the system silently. These jobs do not send a signal to their parent on termination, and `waitjob` provides no information about these jobs.

See `getjtab`(2) for a description of the `jtab` structure.

NOTES

Any process that does not ignore `SIGCLD` signals (see `signal`(2)) and uses `waitjob` must first issue a `wait`(2) system call, which gathers the eldest child of the job when the child exits. If `wait`(2) is not issued, the job will continue to exist, with the eldest process of the job existing as a zombie process; `waitjob` will not consider the job to be terminated.

RETURN VALUES

If `waitjob` completes successfully, the job ID of the terminated job is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `waitjob` system call fails if one of the following error conditions occurs:

Error Code	Description
EAGAIN	The calling process is the parent of one or more jobs configured to send a signal on termination, but none of the child jobs has terminated.
ECHILD	The calling process does not have any child jobs configured to send a signal on termination.
EFAULT	The *jtab* argument points outside the allocated process address space.

SEE ALSO

`getjtab`(2), `setjob`(2), `signal`(2), `wait`(2)

NAME

`wracct` – Writes an accounting record to the kernel accounting file or to a daemon accounting file

SYNOPSIS

`#include <acct/dacct.h>`

`int wracct (char *`*buf*`, int` *did*`, int` *jid*`, int` *nbyte*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `wracct` system call writes an accounting record to a daemon accounting file. If a user enables job accounting, the accounting record will also be written to the user's job accounting file. The `ja`(1) command can process this file.

The `wracct` system call accepts the following arguments:

buf Points to the accounting record. The size (in bytes) of this buffer is specified by *nbyte*. The accounting records are defined in `acct`(5) and in `/usr/include/acct/dacct.h`.

did Specifies the type of accounting record that will be written. These daemon identifiers are specified in `/usr/include/sys/accthdr.h`.

jid Specifies the job ID of the process for which the record is being written. This is usually the job ID contained in the accounting record to which *buf* points.

nbyte Specifies the size (in bytes) of *buf*.

The daemons and the accounting subsystem must enable the appropriate type of accounting by using the `turnacct`(8) or `turndacct`(8) command.

Only a process with appropriate privilege can use this system call.

NOTES

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_ACCT`	The process is allowed to use this system call.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to use this system call.

RETURN VALUES

If `wracct` completes successfully, a value of 0 is returned; otherwise, a value of −1 is returned, and `errno` is set to indicate the error.

ERRORS

The `wracct` system call fails if one of the following error conditions occurs:

Error Code	**Description**
`EINVAL`	An argument that is not valid was passed to the system call.
`EPERM`	The process does not have appropriate privilege to use this system call.

FILES

`/usr/include/acct/dacct.h`	Defines daemon accounting files
`/usr/include/sys/accthdr.h`	Specifies daemon identifiers

SEE ALSO

`jacct`(2)

`ja`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`acct`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`qmgr`(8), `tpdaemon`(8), `turnacct`(8), `turndacct`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`write` – Writes on a file

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

ssize_t write (int fildes, const void *buf, size_t nbyte);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `write` system call writes from a buffer to a file. It accepts the following arguments:

fildes Specifies the file descriptor. It is obtained from an `accept`(2), `creat`(2), `dup`(2), `fcntl`(2), `open`(2), `pipe`(2), `socket`(2), or `socketpair`(2) system call

buf Points to the buffer in which the data is stored.

nbyte Specifies the number of bytes to be written.

On devices capable of seeking, the writing of data proceeds from the position in the file indicated by the file pointer. On return from `write`, the file pointer is incremented by the number of bytes written.

On devices incapable of seeking, writing starts at the current position. The value of a file pointer associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file pointer is set to the end of the file before each write.

If the file being written is a pipe (or FIFO special file), some special semantics apply:

- If the `O_NDELAY` and `O_NONBLOCK` flags in the file flag word are both clear (the normal case), the write request will block until there is room to copy all the data into the pipe.
- If the `O_NDELAY` flag is set (no delay), the number of bytes to be written to the pipe is less than or equal to the value `PIPE_BUF`, and insufficient space exists in the pipe, `write` returns a value of 0 immediately (no blocking) with no data written to the pipe.
- If the `O_NONBLOCK` flag is set (no delay), the number of bytes to be written to the pipe is less than or equal to the value `PIPE_BUF`, and insufficient space exists in the pipe, `write` returns a value of –1 immediately (no blocking) with no data written to the pipe.

- If the O_NDELAY flag is set (no delay), the number of bytes to be written to the pipe is greater than the value PIPE_BUF, and insufficient space exists in the pipe, write copies as many bytes to the pipe as possible and returns the number of bytes written.
- If the O_NONBLOCK flag is set (no delay), the number of bytes to be written to the pipe is greater than the value PIPE_BUF, and insufficient space exists in the pipe, write copies as many bytes to the pipe as possible and returns a value of −1 to the user. (The user is not able to determine the number of bytes actually delivered to the pipe.)

The value PIPE_BUF is defined in the header limits.h and typically has a value of 512 words (4096 bytes).

For regular files, if the O_SYNC flag of the file status flags is set, the write does not return until both the file data and file status are updated physically. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write does not return until the data is updated physically. A write to a regular file is blocked if mandatory file and record locking is set (see chmod(2)) and a record lock is owned by another process on the segment of the file to be written. If O_NDELAY and O_NONBLOCK are both clear the write sleeps until the blocking record lock is removed.

NOTES

A process must be granted write permission to the file via the security label. That is, the active security label of the process must be equal to the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
PRIV_MAC_WRITE	The process is granted write permission to the file via the security label.

If the PRIV_SU configuration option is enabled, the super user is granted write permission to the file via the security label.

RETURN VALUES

If write completes successfully, the number of bytes actually written is returned; otherwise, a value of −1 is returned, and errno is set to indicate the error.

ERRORS

The write system call fails and the file pointer remains unchanged if one of the following error conditions occurs:

Error Code	**Description**
EAGAIN	Mandatory file and record locking was set, O_NDELAY was set, and a blocking record lock exists.
EBADF	The *fildes* argument is not a valid file descriptor open for writing.
EBADF	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.

`EDEADLK` The write was going to go to sleep and cause a deadlock situation to occur.

`EFAULT` The *buf* argument points outside the allocated process address space.

`EFBIG` An attempt was made to write a file that exceeds the file size limit or the maximum file size of the process. See `ulimit`(2).

`EINTR` A signal was caught during the `write` system call (see `signal`(2)).

`ENOLCK` The system record lock table was full; therefore, the write could not go to sleep until the blocking record lock was removed.

`ENOSPC` During a write to an ordinary file, no free space was found in the file system.

`EPIPE` and `SIGPIPE` signals
An attempt is made to write to a pipe that is not open for reading by any process.

`EQACT` A file or inode quota limit was reached for the current account ID.

`EQGRP` A file or inode quota limit was reached for the current group ID.

`EQUSR` A file or inode quota limit was reached for the current user ID.

EXAMPLES

The following examples illustrate different uses of the `write` system call.

Example 1: In this example, the `read`(2) and `write` system calls sequentially update the records of file `datafile`. For each iteration of the *while* loop, a record is read into user memory, updated, and then written back to `datafile`.

A value 0 returned by `read`(2) indicates an end-of-file (EOF) condition has been reached. The data read and written is staged in the system buffer cache because the `O_RAW` flag is not specified on the `open`(2) request.

```
#include <unistd.h>

main()
{
     int fd, cnt;
     char buf[100];

     if ((fd = open("datafile", O_RDWR)) == -1) {
          perror("Opening file datafile failed");
          exit(1);
     }

     while ((cnt = read(fd, buf, 100)) != 0) { /* read returning 0 means EOF */

          /* update data (cnt bytes) in buf here and then write back */

          lseek(fd, (long) cnt, 1);        /* backup to beginning of record */
          if (write(fd, buf, cnt) == -1) { /* write record back to file */
               perror("write failed");
               exit(1);
          }
     }

     printf("EOF reached on file datafile.\n");
}
```

Example 2: In this example, the `read`(2) and `write` system calls perform a simple file copy operation. The first argument to the program is the file name of the file to be copied. The second argument is the file name of the duplicate copy.

```
#include <fcntl.h>

#define BUFSIZE 4096

main(int argc, char *argv[])
{
     int ifd, ofd, noread, nowrite, cnt;
     char buf[BUFSIZE];

     if ((ifd = open(argv[1], O_RDONLY)) == -1) {
          perror("opening input file failed");
          exit(1);
     }

     if ((ofd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1) {
          perror("opening output file failed");
          exit(1);
     }

     while ((noread = read(ifd, buf, BUFSIZE)) != 0) {
          if (noread == -1) {
               perror("read error");
               exit(1);
          }
          cnt = 0;
          do {
               if ((nowrite = write(ofd, &buf[cnt], noread - cnt)) == -1) {
                    perror("write error");
                    exit(1);
               }
               cnt += nowrite;
          } while (cnt < noread);
     }

     close(ifd); close(ofd);
}
```

FILES

`/usr/include/sys/types.h`	Contains types required by ANSI X3J11
`/usr/include/unistd.h`	Contains C prototype for the `write` system call

SEE ALSO

`accept`(2), `chmod`(2), `creat`(2), `dup`(2), `fcntl`(2), `lseek`(2), `open`(2), `pipe`(2), `read`(2), `signal`(2), `socket`(2), `socketpair`(2), `ulimit`(2), `writea`(2)

NAME

`writea` – Performs asynchronous write on a file

SYNOPSIS

```
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>
```

`int writea (int` *fildes*`, char *`*buf*`, unsigned` *nbyte*`, struct iosw *`*status*`,`
`int` *signo*`);`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `writea` system call performs an asynchronous write of a specified number of bytes from a buffer to a file. The first three arguments of the `writea` system call are the same as the `write`(2) system call. The last two arguments are used for I/O completion notification as in the `reada`(2) system call.

The `writea` system call accepts the following arguments:

fildes Specifies a file descriptor. It is obtained from a `creat`(2), `dup`(2), `fcntl`(2), `open`(2), or `pipe`(2) system call or socket descriptor obtained from a call to the `socket`(2) system call

buf Points to the buffer in which the data is stored.

nbyte Specifies the number of bytes to be written.

status Points to a `iosw` structure. This structure is defined in the `usr/include/sys/iosw.h` file.

signo Specifies the signal that should be sent to indicate that the I/O transfer is complete. For a list of signals, see the `signal`(2) man page.

A write to a regular file is blocked if mandatory file and record locking is set (see the `chmod`(2) man page), and a record lock is owned by another process on the segment of the file to be written. If `O_NDELAY` and `O_NONBLOCK` are both clear, the write sleeps until the blocking record lock is removed.

NOTES

A process must be granted write permission to the file via the security label. That is, the active security label of the proces must be equal to the security label of the file.

A process with the effective privilege shown is granted the following ability:

Privilege	**Description**
`PRIV_MAC_WRITE`	The process is granted write permission to the file via the security label.

If the `PRIV_SU` configuration option is enabled, the super user is granted write permission to the file via the security label.

RETURN VALUES

If `writea` completes successfully, the number of bytes remaining to be written is returned; otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

ERRORS

The `writea` system call fails and the file pointer remains unchanged if one of the following error conditions occurs:

Error Code	**Description**
`EAGAIN`	Mandatory file and record locking was set, `O_NDELAY` was set, and a blocking record lock exists.
`EBADF`	The *fildes* argument is not a valid file descriptor open for writing.
`EBADF`	The active security label of the process does not equal the security label of the file, and the process does not have appropriate privilege.
`EDEADLK`	The write was going to go to sleep and cause a deadlock situation to occur.
`EFAULT`	The *buf* or *status* argument is not fully contained in the process address space.
`EFBIG`	An attempt was made to write a file that exceeds the file size limit or the maximum file size of the process. See the `ulimit`(2) man page.
`EINTR`	The process caught a signal during the `writea` system call (see `signal`(2)).
`EINVAL`	The *signo* argument is not a valid signal number or 0.
`ENOLCK`	The system record lock table was full; therefore, the write could not go to sleep until the blocking record lock was removed.
`ENOSPC`	During a write to an ordinary file, no free space was found in the file system.
`EPIPE` and `SIGPIPE` signals	An attempt is made to write to a pipe that is not open for reading by any process.
`EQACT`	A file or inode quota limit was reached for the current account ID.
`EQGRP`	A file or inode quota limit was reached for the current group ID.
`EQUSR`	A file or inode quota limit was reached for the current user ID.

FORTRAN EXTENSIONS

The `writea` system call can be called from Fortran as a function:

```
INTEGER fildes, buf(n), nbyte, status, signo, WRITEA, I
I = WRITEA (fildes, buf(n), nbyte, status, signo)
```

EXAMPLES

The following examples illustrate different uses of the `writea` system call. In each example, the write operation completes in parallel with other work in the user's process. Simpler solutions appear in the last two examples, which make use of additional calls.

Example 1: In this program, the `writea` request specifies the delivery of a `SIGUSR1` signal on the completion of the request.

The program uses the `pause`(2) request to wait for the completion of the asynchronous write operation (that is, reception of the `SIGUSR1` signal). The library routine `sigoff`(3C) provides assurance that the `SIGUSR1` signal is not received before reaching the `pause`(2) request.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

struct iosw wrstat;

main()
{
     char buf[4096];
     int fd;
     void wrhdlr(int signo);

     signal(SIGUSR1, wrhdlr);

     if ((fd = open("newfile", O_WRONLY | O_CREAT | O_RAW, 0644)) == -1) {
          perror("open (newfile) failed");
          exit(1);
     }

     /* Program populates buffer buf with data here. */

     sigoff();      /* delay signal reception until pause() is reached */
     writea(fd, buf, 4096, &wrstat, SIGUSR1);  /* SIGUSR1 sent when
                                                  write completes */

     /* Perform other work here in parallel with I/O completion. */

     pause();       /* wait for write to complete - pause() calls sigon() */

     /* Output data has now vacated buffer buf due to writea. */
}

void wrhdlr(int signo)
{
```

```
          signal(signo, wrhdlr);
          printf("writea wrote %d bytes\n", wrstat.sw_count);
          wrstat.sw_flag = 0;
     }
```

Example 2: (Some system calls in the example are not supported on Cray MPP systems.) Unlike the program in example 1, this program uses the `recalla`(2) system call to wait for completion of the asynchronous output operation. The user's program is informed of the completion by reception of the `SIGUSR1` signal. While `recalla`(2) can wait for the completion of multiple asynchronous I/O requests from multiple files, it only waits for one write operation in this example.

```
     #include <fcntl.h>
     #include <signal.h>
     #include <sys/types.h>
     #include <sys/iosw.h>
     #include <sys/param.h>

     struct iosw wrstat;

     main()
     {
          char buf[4096];
          int fd;
          long mask[RECALL_SIZEOF];
          void wrhdlr(int signo);

          signal(SIGUSR1, wrhdlr);

          if ((fd = open("newfile", O_WRONLY | O_CREAT | O_RAW, 0644)) == -1) {
               perror("open (newfile) failed");
               exit(1);
          }

          /* Program populates buffer buf with data here. */

          RECALL_SET(mask, fd);    /* set bit for fd in mask */

          writea(fd, buf, 4096, &wrstat, SIGUSR1);  /* SIGUSR1 sent when
                                                       write completes */

          /* Perform other work here in parallel with I/O completion. */

          recalla(mask);        /* wait for write to complete */

          /* Output data has now vacated buffer buf due to writea. */
     }

     void wrhdlr(int signo)
```

```
{
     signal(signo, wrhdlr);
     printf("writea wrote %d bytes\n", wrstat.sw_count);
     wrstat.sw_flag = 0;
}
```

Example 3: Unlike the programs in examples 1 and 2, this program does not have an I/O completion signal specified on the `writea` request. The program uses the `recall`(2) system call to wait for the completion of the asynchronous write operation. While `recall`(2) can wait for completion of multiple asynchronous I/O requests from multiple files or even the same file, it only waits for one asynchronous write operation in this example.

```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/iosw.h>

main()
{
     char buf[4096];
     int fd;
     struct iosw wrstat[1], *statlist[1];

     if ((fd = open("newfile", O_WRONLY | O_CREAT | O_RAW, 0644)) == -1) {
          perror("open (newfile) failed");
          exit(1);
     }

     /* Program populates buffer buf with data here. */

     writea(fd, buf, 4096, &wrstat[0], 0); /* no signal sent when
                                              write completes */
     statlist[0] = &wrstat[0];

     /* Perform other work here in parallel with I/O completion. */

     recall(fd, 1, statlist);               /* wait for write to complete */

     printf("writea wrote %d bytes\n", wrstat[0].sw_count);
     wrstat[0].sw_flag = 0;

     /* Output data has now vacated buffer buf due to writea. */
}
```

SEE ALSO

`chmod`(2), `creat`(2), `dup`(2), `fcntl`(2), `open`(2), `pause`(2), `pipe`(2), `reada`(2), `recall`(2), `recalla`(2), `signal`(2), `socket`(2), `ulimit`(2), `write`(2)

`sigoff`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080