Introductory Material

Examining Kernel Code

This section describes the structure of the UNICOS kernel source code, and gives some tips on how to examine the code for debugging purposes.

Structure of kernel source code

In the released system, the kernel source code is in the directory **/usr/src/uts**. (This directory should not be accessible to users.) This manual describes the source code in this directory and its subdirectories. Associated definition files (*include files*, also called *header files*) are also described; they are found in the source directory **/usr/include/sys**.

This manual does not describe the software in other directories (for example, the TCP/IP source code).

The **uts** directory contains the following subdirectories:

Subdirectory	Description
boot	Boot subdirectory
cf	Configuration subdirectory
fs	File system subdirectory
io	I/O subdirectory
md	Machine-dependent subdirectory
os	Operating system subdirectory

The directory **uts** also contains files used in building the kernel; for example, the file **osdef.s** is the kernel's version of the assembler definition file used in building the kernel.

Figure 1 shows the structure of the **/usr/src/uts** directory.

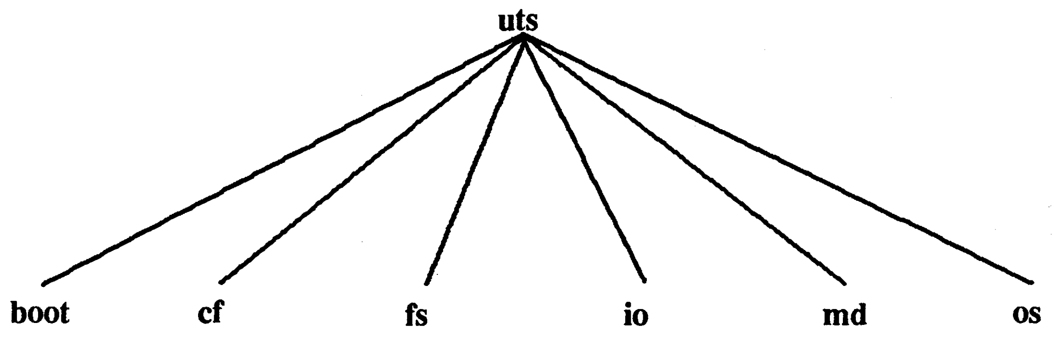

Figure 1. Structure of the **uts** directory

The directory **/usr/include/sys** is the system include file subdirectory. It contains the include files needed to build the kernel.

The remainder of this subsection describes the contents of the subdirectories.

The boot subdirectory

This subdirectory contains the source files **boot.c**, **booth.s**, and **boot.mk**. It also contains the UNICOS boot program, **osboot**, if that program has been built.

The cf subdirectory

This subdirectory contains the following files and subdirectories:

File	Description
Makefile	Kernel makefile.
conf.c	Configuration file.
name.c	File that contains the date and time of the kernel generation, as well as system information returned

File	Description
	by the **uname**(2) system call. **name.c** is edited at kernel compilation time to reflect the **uname** information; this editing is done in the kernel makefile.
hconf.h	Hardware configuration file; used for configuring devices for the foreground processor.
lib	Subdirectory containing UNICOS object files used in building the kernel.
libtcp	Subdirectory containing TCP/IP object files used in building the kernel.
libnfs	Subdirectory containing UNICOS NFS object files used in building the kernel.

For more information on these files, or on system configuration in general, see the *UNICOS System Administrator's Guide for CRAY-2 Computer Systems*, publication SG-2019.

The fs subdirectory

This subdirectory contains the file-system-specific portion of the file system switch (FSS) in the directories **c2**, **proc**, and **sl**.

The directory **c2** contains the file system routines for the native file system (**C2FS**).

The directory **proc** contains the file system routines for the **/proc** file system.

The directory **sl** contains the file system routines for the SUPERLINK file system (**SLFS**). (Only the kernel-level code for SUPERLINK is in the UNICOS 5.0 release; customers wishing to obtain SUPERLINK must order it as a separate product.)

See "File System Switch," page 53, for more information on these file systems.

The io subdirectory

This subdirectory contains the UNICOS device drivers. For example, device drivers for the disks, the HYPERchannel, and the CPUs are located in this subdirectory.

The md subdirectory

This subdirectory contains the machine-dependent code in the UNICOS kernel; this code includes routines for context switching, error processing, and interrupt handling. All of the kernel assembly language files are in this subdirectory.

The os subdirectory

This subdirectory contains the routines that are the core of the UNICOS operating system.

The /usr/include/sys subdirectory

This directory contains the include files that the UNICOS kernel uses when it is being built.

Kernel data structures

The *kernel data structures*, also referred to as *tables*, hold important information for the kernel. For example, the process table is an array that holds information about active processes on the system. The "Kernel Data Structures" section, page 15, describes some important data structures in UNICOS.

Tips for examining kernel code

This subsection describes the use of the **ctags**(1) command with the **tag** feature of the **ex**(1) and **vi**(1) editors to search for kernel routines.

Creating an index of kernel routines

To print an index of kernel routines, listing the routine name, the location of the definition, and the line number of the definition, perform the following operations:

- Change to a directory for which you have write permission (for example, **/tmp**).
- Use the **ctags**(1) command to create a tag file for the kernel directories in **/usr/src/uts**, as follows:

  ```
  ctags -x /usr/src/uts/cf/* \
           /usr/src/uts/fs/c2/* \
           /usr/src/uts/fs/proc/* \
           /usr/src/uts/fs/sl/* \
           /usr/src/uts/io/* \
           /usr/src/uts/md/* \
           /usr/src/uts/os/* \
           /usr/include/sys/* \
           /usr/include/sys/fs/* > tag
  ```

  This creates a file called **tag**; it contains a list of routine names and the full path names of the files containing those routines.
- Print this file; use landscape mode or a wide printer, if possible, as some of the lines are very long.

Creating a tag file for ex or vi

To provide a functional index capability for the **ex** and **vi** editors, use the **ctags** command as follows:

- Change to a directory for which you have write permission (for example, **/tmp**).
- Use the **ctags** command to create a tag file for the kernel directories in **/usr/src/uts**, as follows:

  ```
  ctags  /usr/src/uts/cf/* \
         /usr/src/uts/fs/c2/*  \
         /usr/src/uts/fs/proc/*  \
         /usr/src/uts/fs/sl/*  \
         /usr/src/uts/io/*  \
         /usr/src/uts/md/*  \
         /usr/src/uts/os/*  \
         /usr/include/sys/* \
         /usr/include/sys/fs/*
  ```

  You may wish to redirect standard error output, as this command issues several screenfuls of warning messages.

  This creates a file called **tags**; it contains a list of routine names, the full path names of the files containing those routines, and **ex** (or **vi**) directives for locating those routines.
- Move this file to the directory from which you will examine kernel code (for example, your home directory or **/usr/src/uts**). The **ex** and **vi** commands look for a tag file called **tags** in the current directory.

To use this tag file, type an **ex** or **vi** command with the following format:

```
ex –ta routine
```

or

```
vi –ta routine
```

For example, to see the routine **copen()** without having to specify the file it is in, type the following command:

```
vi –ta copen
```

Once you are in the **ex** or **vi** editor, use the **ta** command to locate another routine in the same or a different file, as in the following example:

```
:ta  chdir
```

For more information, see the **ctags**(1), **ex**(1), and **vi**(1) man pages.

Kernel Data Structures

UNICOS source files and include files define some important tables, maps, linked lists, caches, areas, buffers, pools, and queues. This section describes these entities, using the term *data structures* to refer to them all.

This section is a "mini-manual" by itself (similar to the CRI tables manuals for COS). That is, the discussion of data structures is a discussion of the basic structure and organization of the UNICOS kernel as a whole. This section presents a summary of important points, with many details excluded. Where possible, references to more complete material are included.

The data structures are described in terms of the data they hold, not how the data is assigned or manipulated. This information can be helpful when trying to solve kernel problems (dump busting).

callout – The callout table

The callout table, **callout** (also called the *time-out table*), is used to call a routine after a given amount of time. This table is checked on every clock interrupt (every *clock tick*), and the routines are executed when the specified delay time has expired. Each entry in **callout** specifies the address of a routine to be invoked, a parameter for that routine, and the real-time clock value when the routine should be called. The entries are maintained in chronological order.

The callout table is an array of **callo** structures; the **callo** structure is defined in the include file **sys/callo.h**. The size of the callout table is determined by the **NCALL** parameter in **sys/param.h**.

coremap – The memory map

The memory map, **coremap**, is used to keep track of free memory. Each entry in **coremap** defines a segment of main memory available for allocation. An entry is composed of an address and a size in 512-word clicks. The **malloc**() and **mfree**() routines manage **coremap**, allocating and freeing memory, respectively. At system initialization time, one entry in **coremap** defines all memory available to the user.

coremap is described by the **map** structure, which is defined in the include file **sys/map.h**. The size of **coremap** is determined by the **CMAPSIZE** parameter defined in **sys/param.h**.

dblock – The dynamic block

A dynamic block holds volatile information about a given file system; this includes the list of free sectors, the free i-node bitmap, the free track bitmap, total free blocks, total free tracks, and total free i-nodes. The dynamic block is read into the system buffers when the file system is mounted. A pointer to the dynamic block for a file system is kept in the mount table.

The **dblock** structure describes a dynamic block for CRAY-2 file systems; **dblock** is defined in the include file **sys/fs/c2filsys.h**.

See "UNICOS File System," page 47, for more information on the dynamic block.

devblock – The device control block

The device control block, **devblock**, is used along with the **fcpublk** table for communication between the foreground processor (FP) and the UNICOS kernel. **devblock** contains an entry for each device configured in the FP. This includes the clock, console input, console output, each physical disk, each striped disk device, each HYPERchannel input and output channel, each low-speed channel, each extended low-speed channel, and each background processor. Each entry contains fields for an FP parameter and reply word, the FP device numbers, and the address of the interrupt handler routine.

devblock is created by **md/lowcm.s**; its configuration must match that of the FP system. **devblock** is loaded at a specific address (0200) in low memory; this address is referenced by the FP system.

devblock is an array made up of **devdata** structures; it is indexed by the FP index number. The **devdata** structure is defined in the include file **sys/devdata.h**.

See "Foreground Processor and Kernel Communication," page 33, for more information on **devblock**.

diskspec – The disk specification table

The disk specification table, **diskspec**, is used to hold information about the physical location of a given partition. The data in the **diskspec** table is copied from the device i-node when the device is opened. In addition to partition addresses, the **diskspec** table has fields that specify the type of disk, the number of sectors per track, the number of tracks per cylinder, and the number of striped devices per striped group.

The **diskspec** table is an array of **diskspec** structures indexed by the minor device number. The **diskspec** structure is defined in the include file **sys/diskspec.h**. The maximum number of entries in **diskspec** is determined by the parameter **NMNT** in the include file **sys/param.h**.

dskutab – The disk unit table

The disk unit table, **dskutab**, contains I/O control information for each physical disk device defined on the system. The disk unit table is used only by the disk driver and contains information pertaining to the currently active I/O operation. This table contains an entry corresponding to each physical disk device, plus one for each striped device. Each entry maintains pointers to the head of a linked list of buffers queued for I/O.

The **dskutab** table is an array of **iobuf** structures indexed by the FP index number. The number of usable entries in **dskutab** is determined by the physical disk configuration of the FP in conjunction with the **devblock** table in **md/lowcm.s**. The maximum number of entries in **dskutab** is determined by the parameter **DSKMAX** in the include file **sys/param.h**.

errtab – The error table

The error table, **errtab**, is used to keep track of hardware errors, including single-bit errors, double-bit errors, local memory parity errors, floating-point range errors, and floating-point table errors. Each entry in the error table contains the type of the error, a count of how many times the error has occurred, and an exchange package.

The error table is described by the **errtab** structure, which is defined in the include file **sys/errtab.h**.

fcpublk – The FP/CPU control block

The FP/CPU control block, **fcpublk**, is used along with the **devblock** table to communicate between the background processors and the FP. Each entry in **fcpublk** contains the exchange packets and the FP call word used to communicate CPU requests to the FP.

There is one entry in **fcpublk** for each CPU. The **fcpublk** is loaded at a specific address because it is referenced by the FP system.

fcpublk is an array of **fcpudata** structures indexed by CPU number. The **fcpudata** structure is defined in the include file **sys/cpudata.h**.

See "Foreground Processor and Kernel Communication," page 33, for more information on **fcpublk**.

file – The file table

The file table is used to hold information about open files. Each open file in the system has an entry in the file table; each entry holds specific information about the open file: an offset into the associated file, a status flag (for example, an open-for-reading flag) and a pointer to the corresponding i-node. An entry in the file table is the result of an **open**(2), a **creat**(2), or a **pipe**(2) system call.

The file table is an array of **file** structures. The **file** structure is defined in the include file **sys/file.h**. The size of the file table is determined by the parameter **NFILE** in **sys/param.h**.

Available **file** structures are stored in a linked list called **ffreelist**.

flox – The file lock table

The file lock table, **flox**, is an array holding information about the file locks for a file. There is one entry for each locked region of the file. Each entry includes the following information:

- Type of lock (read or write)
- Start and end of lock (stored as byte offsets)
- Ownership of lock
- Other processes waiting for the lock

The **i_filocks** field in the i-node points to the file locking structure (**filock**) for the associated file. All file locks applied to a given file are threaded together in a linked list connected to the **inode** structure of a given file.

The **filock** structure is defined in the include file **sys/flock.h**. **flox** is defined in **cf/conf.c**. The maximum number of file lock regions (**NFLOCKS**) is defined in **sys/param.h**.

fstypsw – The file system switch table

The file system switch (FSS) table, **fstypsw**, defines all entry points to each file system's routines. It is an array of file-system-specific routines, indexed by the file system type. File-system-independent routines call the file-system-specific routines through this array.

Each i-node contains a pointer to **fstypsw** in the field **i_fstypp**. This pointer is used by the FSS macros (defined in **sys/fstyp.h**) to access the correct file-system-specific routine for each operation.

fstypsw is defined in the include file **sys/conf.h** and is initialized in the file **cf/conf.c**.

See "File System Switch," page 53, for more information.

inode – The in-core i-node table

The in-core i-node table, **inode**, is used to hold information on the i-nodes of active files. It contains an entry for each open file.

The **inode** table is an array of **inode** structures. The **inode** structure is defined in the include file **sys/inode.h**; the size of the **inode** table is determined by the parameter **NINODE** in **sys/param.h**.

UNICOS uses two types of doubly-linked lists, **ifreelist** and **hinode**, to maintain the i-node entries.

The linked list **ifreelist** is a list of available i-node table slots. When the reference count of an i-node in an i-node table slot goes to 0, the slot is linked to the **ifreelist**. At boot time, all the i-node entries are linked into this list.

The linked list **hinode** is the i-node hash table; each entry is a linked list of allocated i-node table slots. To get the index of the start of the appropriate hash list, UNICOS uses a hashing algorithm on the device number and i-node number (*i-number*).

ioh – The I/O header table

The I/O header table, **ioh**, is used to keep track of I/O requests. An I/O header is allocated for every I/O request (read or write operation). The request is broken up into one or more atomic units for the actual I/O. Each atom has one or more corresponding **buf** or **pbuf** structures allocated; these structures point back to the I/O header.

The **io_addr** field of the I/O header is set to the beginning of the first atom, and the **io_endaddr** field is set to the end of the last atom. If an I/O error occurs in an atom, and its address field is less than **io_endaddr**, the field **io_error** is set to the error code, and **io_endaddr** is set to the atom's beginning address.

When all atoms have completed, the difference between **io_endaddr** and **io_addr** is the amount of data successfully transferred.

The I/O header is described by the **ioh** structure, which is defined in the include file **sys/ioh.h**.

jtab – The job table

The job table, **jtab**, is used to keep track of active jobs in the system. There is one entry in the job table for each active job.

The job table is an array of **jtab** structures; the **jtab** structure is defined in the include file **sys/jtab.h**.

kcpublk – The kernel control block

The kernel control block, **kcpublk**, holds per-CPU information for the kernel.

kcpublk is an array of **kcpudata** structures indexed by CPU number. The **kcpudata** structure is defined in the include file **sys/cpudata.h**.

lnode – The limits node table

The limits node table, **lnode**, contains per-user process information for the Share scheduler. Each active user on the system has an entry in the **lnode** table. Each entry in the process table (each **proc** structure) contains a pointer (**p_lnode**) to the **lnode** table.

The **lnode** table consists of **lnode** structures; the **lnode** structure is defined in **sys/lnode.h**.

mount – The mount table

The mount table, **mount**, is used to keep track of mounted file systems. Each entry contains the pointers to i-node structures necessary to link the root of the mounted file system to the directory it is mounted on. A **mount** table entry also contains a pointer to the file-system-specific **mount** structure. (In UNIX systems, this pointer points to the in-core super block instead of to a **mount** structure.)

The table is an array of **mount** structures. The **mount** structure is defined in the include file **sys/mount.h**. The size of the **mount** table is determined by the **NMOUNT** parameter in the include file **sys/param.h**.

pipchan – The pipe channel table

The pipe channel table, **pipchan**, contains information about the pipes in the system. There is one entry for each open pipe; the entry includes the number of readers of the pipe, the number of writers of the pipe, and data buffers in use.

The **pipchan** table is defined in the include file **sys/pip.h**.

proc – The process table

The process table contains information about each process on the system; this information includes the process status and priority, and the address of the **user** structure.

The process table is an array of **proc** structures; one **proc** structure is allocated per active process. The **proc** structure is defined in the include file **sys/proc.h**. The number of process table entries is determined by the **NPROC** parameter in **sys/param.h**.

Unless the associated process is currently executing in a CPU, each **proc** structure is linked into one of two lists: the run queue or the sleep queue. Part of the **proc** structure is the process common structure, **pcomm**. There is one **pcomm** structure active for each multitasked process group. (Every process is in a multitasked group, even if the group contains only one process.)

resinfo – The checkpoint/restart table

The checkpoint/restart table, **resinfo**, is a free list of restart info buffers used to hold dynamic information during checkpoint and restart operations.

The **resinfo** structure is defined in **sys/restartinf.h**; the **resinfo** table is defined and initialized in **os/restart.c**.

slg_buf – The security log buffer

The security log buffer, **slg_buf**, is used to buffer security log messages generated by the kernel until these messages are read and recorded by the security log daemon. **slg_buf** includes the buffer that the security log routines use to store security-related events, input and output pointers into the buffer, and flags describing the buffer's status.

Each entry in the security log buffer is described by the structure **slg**, which is defined in the include file **sys/slog.h**. The security log buffer is initialized by the routine **slginit()** in the module **os/slogext.c**.

statblk – The device index block

The device index block, **statblk**, is an area in low memory (0100) used by the kernel and FP as an index to the devices configured in the system. The **statblk** contains information on the size of physical memory, the size of the **devblock**, and the base address and index number for each device.

The **statblk** is created from information in **md/lowcm.h** and **md/lowcm.s**. See "Memory Layout," page 39, for more information on the **statblk**.

stripeblk – The stripe block

The stripe block (**stripeblk**) is an area in low memory that is used to map stripe device references to physical disks. The CAL routines in **md/lowcm.s** set up the stripe block beginning at location 060 (octal).

The stripe block contains up to 4 entries, one for each logical stripe device configured. The entries are located at 060, 064, 070, and 074. Each entry is an array of 32 bytes (4 words). Each byte contains the FP index number of one of the members of this stripe device. Unused entries in **stripeblk** are filled with 0's.

The macros that **md/lowcm.s** uses to define the stripe block are located in **md/lowcm.h**.

swap_bb – The swap bad block table

The swap bad block table, **swap_bb**, contains a list of bad (flawed) blocks residing on the swap device. The bad block information is copied from the swap super block at system initialization time.

The **swap()** routine scans **swap_bb** for bad blocks before every I/O operation and then avoids the bad blocks.

The **swap_bb** table is declared in **cf/conf.c**. The number of entries is defined by the expression **NSWBAD + NSWFX * FXNBAD** (defined in the include file **sys/fs/c2filsys.h**). The swap super block is described by the **swblock** structure, which is defined in **sys/fs/c2filsys.h**.

swapmap – The swap map

The swap map, **swapmap**, is a map of free swap space. Each entry in **swapmap** defines a segment of the swap space available for allocation. An address and a size in 512-word blocks make up each entry.

The **malloc()** and **mfree()** routines manage the swap map, allocating and freeing swap space, respectively. At system initialization time, one entry in **swapmap** defines all available swap space.

swapmap is an array of **map** structures; the **map** structure is defined in the include file **sys/map.h**. The size of **swapmap** is determined by the **SMAPSIZE** parameter defined in **sys/param.h**.

sysent – The system call entry point table

The system call entry point table, **sysent**, is used to map system call numbers to kernel routines.

The **sysent** table is defined in **os/sysent.c**.

See "System Calls," page 65, for more information on the **sysent** table.

System buffer cache

The system buffer cache is used to hold, or *buffer*, data between users and devices for buffered I/O operations. The data resides in the buffer for as long as possible, thereby reducing the number of device I/O calls for subsequent reuse of data.

The system has the following types of buffer pools:

- A sector buffer pool (for sector I/O)
- A track buffer pool for each type of disk device configured in the system; that is, a pool for track I/O on DD-29 drives, a pool for track I/O on DD-40 drives, and a pool for track I/O on DD-49 drives. There is also an optional track buffer pool for each type of striped device configured in the system; that is, a pool for striped DD-29 drives and a pool for striped DD-49 drives.
- A physical buffer pool for unbuffered, or *raw*, I/O. The buffer headers are maintained the same way as for buffered I/O, except there are no data buffers associated with them.

Each buffer pool consists of an array of **buf** structures called *buffer headers*. Buffer headers are allocated at system initialization time; size information is located in the **sysvars** table in **cf/conf.c**. Each buffer header has all the information needed for doing the I/O, including the device number, block number, byte count, and completion status; it also has a pointer to an associated I/O data buffer.

Each buffer header may be linked into two lists; it is always linked to the hash table (hashed on the device and block number of the device with which it is currently associated) and is linked to the available list when not in use.

The **buf** structure is defined in the include file **sys/buf.h**.

The trace buffer

The trace buffer is a a circular buffer that contains the entries written by the operating system trace macro **UTRACE**.

The trace buffer is allocated in the file **md/trace.s**. The symbol **etbegin** marks the first entry in the trace buffer.

See "Trace Buffer," page 149, for more information on the trace buffer.

user – The user structure

The **user** structure (also called the *user area*) holds information about a process. One **user** structure is allocated for each active process. Unlike the **proc** structure, the **user** structure is needed only when the process is running. Because of this, the **user** structure is swapped with the process.

The **user** structure contains a register save area, I/O control and status information, and the per-process system stack.

The **user** structure is defined in the include file **sys/user.h**. Memory space for **user** structures is allocated as needed.

ucomm – The user common structure

The user common structure, **ucomm**, contains information common to groups of processes. There is one **ucomm** structure for each multitasked process group. The **ucomm** structure includes the user's user ID and group ID, pointers to file table entries of open files, and accounting information.

The **ucomm** structure is located at the beginning of the process' memory image or swap image. The **pcomm** structure contains the address of the image.

The **ucomm** structure is defined in the include file **sys/user.h**. Memory space for **ucomm** structures is allocated as needed.

System Initialization

System initialization for UNICOS running on CRAY-2 systems is performed from the system control console (SCC). The SCC deadstarts the foreground processor (FP), then loads and begins execution of UNICOS in the background processor (BP). This initialization is performed by an SCC command script, **boot**.

This section describes the boot procedure (both slow and fast boot operations) and provides an overview of UNICOS initialization.

Boot procedure

SCC disk storage contains the SCC command script **boot** and the files containing the FP programs and the UNICOS kernel binary.

The **boot** script performs the following steps:

1. Loads the BP reciprocal and square-root approximation tables from the file **ratable**.
2. Deadstarts the FP program **dumper**; this program allows the SCC to write to FP local memory.
3. Deadstarts the FP system support program **fpsys**. This includes copying the FP local memory image from the file **fpsys** to FP local memory.
4. Copies the UNICOS kernel binary from the SCC file **unicos** to common memory.
5. Starts the BP with the exchange package in the kernel.

The SCC **boot** script also sets displays for the SCC. In particular, the system console display is set; the SCC then waits for further initialization from the UNICOS kernel.

Fast boot procedure

Copying the UNICOS kernel binary to common memory takes several minutes on a slow SCC. To speed up this process, the fast boot procedure may be used with a configured UNICOS system. The following operations are necessary:

1. Place the UNICOS kernel on a dedicated disk partition (**/dev/dsk/os**).
2. Compile the program **osboot** with the definition for **/dev/dsk/os**.
3. Copy **osboot** to the SCC disk.
4. Modify the SCC **boot** script to copy **osboot** to common memory and start the BP with the **osboot** exchange package.

Once these operations have been performed, **osboot** then copies the kernel from the dedicated disk partition to common memory during the fast boot procedure.

See the *UNICOS System Administrator's Guide for CRAY-2 Computer Systems*, publication SG-2019, for a complete description of the deadstart procedure.

UNICOS initialization

This subsection describes UNICOS initialization on the background processors. It covers the actions of the CAL routines in the module **md/inout.s**, the **main**() routine, and the **init** process.

The routines in md/inout.s

UNICOS execution starts at the entry point **BEGIN***n* (*n* is the CPU number; for example, 00 for CPU A) in **md/inout.s**. This code sets up the initial exchange package (**XASYS**) for the CPU and the initial stack information.

This code performs the following functions:

- Clears local memory
- Sets up a shared system stack area
- Calls **main**()

Control transfers to the **main**() routine (in **os/main.c**).

The main() routine

The **main**() routine (in **os/main.c**) performs the following operations:

- Clears kernel BSS and user memory.
- Initializes the memory map (**coremap**).
- Clears the semaphores.
- Sets the clock from the SCC.
- Sets up the first system process, process 0 (also called **proc[0]**); this process later turns into the swapper.
- Initializes the lnodes (used for Share scheduling).
- Creates process 1 (**proc[1]**, or **init**) and the idle processes. The file **md/icode.s** contains the code executed by the idle processes.
- Prints console startup messages.
- Calls system initialization routines **iinit**(), **secure_init**() (if the system is running in secure mode), and **restartinit**().
- Mounts the root file system.

- Puts **init** on the run queue.
- Calls **iget**() to open the root and swap devices.
- Calls **sched**() (the swapper), which switches to the **init** process. Control never returns to **main**().

Note

If, for some reason, the file **/etc/init** *is missing or corrupted, the system hangs (after printing "Scheduling CPU A"). The lights on the mainframe blink in a regular, alternating pattern. This pattern corresponds to the error number,* **errno,** *resulting from the* **exec** *operation of* **/etc/init.**

The init process

The **init** process (in **/etc/init**) performs the following actions:

- Checks **/etc/inittab** for a default level.
- Sets up the console (opens the console device, **/dev/console**).
- Executes the Bourne shell.
- Executes the script **/etc/brc** to initialize the mount table.
- Initializes the **/etc/utmp** file and enters the time of the boot.

At this point, the system is in single user mode; further actions depend on the operator or system administrator. See the *UNICOS System Administrator's Guide for CRAY-2 Computer Systems*, publication SG-2019, for information on system start procedures.

Foreground Processor and Kernel Communication

This section describes communication between the foreground processor (FP) and the UNICOS kernel. It covers the following topics:

- Common configuration
- The device communication block, **devblock**
- Relative device addressing
- The FP/CPU control block, **fcpublk**

Common configuration

A common configuration file, **cf/hconf.h**, is used to build the FP system and the tables in low common memory that are used for communication between the FP and the UNICOS kernel. Two tables, **devblock** and **fcpublk**, are assembled in the UNICOS kernel module **md/lowcm.s**.

The file **hconf.c** contains an entry for each physical device, each background processor, and any other information that is to be shared by the FP and the UNICOS kernel.

Both the UNICOS kernel module **md/lowcm.s** and the **fpsys** source include the file **hconf.h**, using the CAL *include* feature. **hconf.h** contains CAL macro calls to a separate set of CAL macro definitions. The macro definitions for generating the **lowcm** module are in the file **md/lowcm.h**. The macro definitions for **fpsys** are in the binary definition file **fp/fp.t**. Use of the same configuration file (**hconf.h**) ensures that the FP system and the

UNICOS kernel always agree on the device configuration and the areas of common memory where I/O and control requests are communicated.

For a more detailed description of the device macro parameters, see the *CRAY-2 Foreground Processor Reference Manual*, publication SP-2020.

The device communication block (devblock)

The device communication block, known to UNICOS as **devblock**, is located in common memory starting at address 0200. There is an entry for each physical I/O device, an entry for the console, and an entry for real-time clock interrupts. Some devices may use two entries in the **devblock**, one for input and one for output.

Currently, each entry in **devblock** is 7 words in length. There are a maximum of 64 entries. The structure of a device block entry is defined by the **devdata** structure in the file **sys/devdata.h**, as follows:

```
struct devdata {
        word fgpdata;       /* inout use only          */
        word fgpparam;      /* parameter and reply word*/
        int  (*intr)();     /* interrupt service address */
        uint fgpdev;        /* fgp device number       */
        uint index;         /* fgp index number        */
        uint pdev;          /* physical device number  */
        word aux;           /* aux - driver dependent  */
};
```

The **fgpdata** field receives the FP reply word from **fpsys** upon completion of an I/O request. When UNICOS acknowledges the I/O interrupt, the FP reply word is moved to the **fgpparam** field. The **fgpparam** field is also used to hold the I/O parameter word picked up by **fpsys** on an I/O request.

The **intr** field holds the address of the device interrupt service routine (the *interrupt handler*).

The **fgpdev** field is the index into the **fpsys** internal device tables.

The **index** field specifies an entry's position in the **devblock.**

The **pdev** field is the physical device address; it is for informational use such as error logging. Physical addressing is done in **fpsys** based on the **fgpdev** value.

The **aux** field is device dependent; it normally contains additional configuration information, such as a controller revision level.

For a more detailed description of the use of **devblock** entries for all supported devices, see the *CRAY-2 Foreground Processor Reference Manual*, publication SP-2020.

Relative device addressing

Each entry in the **devblock** is referenced by its FP index number. The FP index number is the absolute location of a given entry in the **devblock.** Devices of a given type are always in contiguous **devblock** entries. For example, **devblock** entries for disk devices may start at FP index 3. The first index number of a given device type is known as its *device base.*

For each type of device, there is a base and a count of the number of devices of that type; these are the **devbase** and **ndev** numbers, respectively. The **devbase** and **ndev** numbers are assembled starting at address 0100 of common memory and are labeled with the ASCII names of the device type to provide for easy reference.

The **devbase** number is used by the driver for that device type to provide a relative addressing mechanism. The **ndev** number tells the driver how many devices of a given type exist.

For example, the disk driver assigns a pointer, **ddp,** to a device's **devblock** location as follows:

```
ddp = &devblock[index+diskbase];
```

index is the relative index number from the device i-node.

Table 1 shows the device **devbase** and device **ndev** variables defined in the module **md/lowcm.s** and used by the device drivers.

Table 1. **devbase** and **ndev** variables defined in **lowcm.s**

devbase	ndev	Description
cpubase	**nbp**	Base for and number of CPUs
diskbase	**ndisk**	Base for and number of disk devices
hsxbase	**nhsx**	Base for and number of high-speed channel (HSX) devices
hybase	**nhy**	Base for and number of NSC A130 HYPERchannel devices
lspbase	**nlsp**	Base for and number of low-speed channel (LSP) devices
lsxbase	**nlsx**	Base for and number of full-duplex low-speed channel (LSX) devices
stripebase	**nstripe**	Base for and number of logical striped devices
tsbase	**nts**	Base for and number of tape subsystem devices

FP/CPU Control Block

The FP/CPU control block, known to UNICOS as **fcpublk,** contains the exchange packages and an FP call word for each background processor. The FP uses the exchange packages in **fcpublk** to control the background processors' interrupt processing and state switching between user and system mode. An FP call word is used to pass I/O requests to the FP.

fcpublk is defined by the **fcpudata** structure in the include file **sys/cpudata.h. fcpublk** currently starts at address 01200 in common memory. The size of an entry corresponding to one

CPU is currently 40 octal words. The **fcpudata** structure is defined as follows:

```
struct  fcpudata {
        word    fgpcall;      /* FP call word                                  */
        word    align1[2];
        word    realtime;
        xp_t    serrxp;       /* XASER - entry for system mode errors          */
        sv_t    serrsv;       /* XASTP - saved during system mode errors       */
        xp_t    osxp;         /* XASYS - entry for most of operating system    */
        sv_t    ossv;         /* XAINOUT - saved during system entrance        */
        xp_t    userxp;       /* XAUSER - copy of packet for current user      */
        xp_t    resxp;        /* system restart xp                             */
        word    align2[4];
};
```

The UNICOS kernel requests service from the FP with the CPU **exit** instruction. The *exit* parameter specified in the **exit** instruction tells the FP what kind of service is requested. For I/O requests, the FP call word has been filled as required by the driver before the **exit** operation.

For a more detailed description of CPU state control, exchange packages, and the use of the FP call word, see the *CRAY-2 Foreground Processor Reference Manual*, publication SP-2020.

Memory Management

Memory Layout

This section describes the layout of common memory and local memory. It covers the following topics:

- Printing memory contents with **crash**(1M) and **nmab**(1)
- Layout of common memory
- Layout of low common memory (also called *low memory*), including an example of **crash** output
- Layout of local memory, including an example of **nmab** output
- Memory management restrictions

Printing memory contents

The layout of memory can change with each system reconfiguration. To see the layout of your system, use **crash**(1M) or **nmab**(1) as in the following examples.

Use the **crash** directive **od**, as in the following example:

```
crash
> od starting_address number_of_words
```

Use the following command format:

```
nmab –nw kernel_binary
```

Following subsections contain examples produced with these commands.

Layout of common memory

The layout of common memory changes with each system reconfiguration. To see the layout, use the **nmab**(1) command as described in the following subsection.

Layout of low memory

The contents of low memory are determined by the following files:

File	Description
cf/hconf.h	Contains definitions of devices and location of devices
md/lowcm.h	Contains macros for device configuration
md/lowcm.s	Contains information to set up low memory

Sites change **hconf.h** as needed, but do not normally change **lowcm.h** and **lowcm.s**. Each time **hconf.h** is changed, the layout of low memory changes.

The remainder of this subsection describes the file **md/lowcm.s** and shows an example of **crash**(1M) output.

File md/lowcm.s

The file **lowcm.s** contains assembly directives to set up low memory. One of its actions is to define symbols for some absolute addresses in low memory; these symbols can be used in

crash to print the contents of interesting device tables. These definitions are as follows:

stripeblk =	zero@cm+o'60
statblk =	zero@cm+o'100
ioactive =	zero@cm+o'170
devblock =	zero@cm+o'200
fcpublk =	zero@cm+o'1200
dumpblk =	zero@cm+o'1400
tsdvblock =	zero@cm+o'1430
cpublzzz =	zero@cm+o'2000

Example: crash(1M) output

This example shows a dump of low memory from the **crash**(1M) command. The first column shows the word number, the second column shows the contents of the word as an octal word, and the third column shows the contents of the word as characters. The rightmost column contains comments, in italics, to aid in identifying the words; this information does not appear in **crash** output.

```
> od 0 100

00000000000:  0000070000000000000000  ........
00000000001:  0000000000000000000000  ........
*
00000000060:  0040160000000000000000  ........
00000000061:  0000000000000000000000  ........
*
00000000100:  0701503627155531266440  physmem
00000000101:  0000000000002000000000  ........
00000000102:  0671443127304010020040  ndev
00000000103:  0000000000000000000054  .......,         size of statblk
00000000104:  0621513466544010020040  disk
00000000105:  0000000000000000000003  ........         diskbase
00000000106:  0000000000000000000051  .......)         ndisk
00000000107:  0715643446456031220040  stripe
00000000110:  0000000000000000000053  .......+         stripebase
00000000111:  0000000000000000000001  ........         nstripe
```

```
00000000112:  0641711002004010020040  hy
00000000113:  0000000000000000000033  ........     hybase
00000000114:  0000000000000000000001  ........     nhy
00000000115:  0661633402004010020040  lsp
00000000116:  0000000000000000000777  ........     lspbase
00000000117:  0000000000000000000000  ........     nlsp
00000000120:  0721633462004010020040  tss
00000000121:  0000000000000000000043  .......#     tsbase
00000000122:  0000000000000000000002  ........     nts
00000000123:  0641633602004010020040  hsx
00000000124:  0000000000000000000047  .......'     hsxbase
00000000125:  0000000000000000000004  ........     nhsx
00000000126:  0661633602004010020040  lsx
00000000127:  0000000000000000000035  ........     lsxbase
00000000130:  0000000000000000000003  ........     nlsx
00000000131:  0615603522004010020040  cpu
00000000132:  0000000000000000000077  .......?     cpubase
00000000133:  0000000000000000000004  ........     nbp
00000000134:  0000000000000000000000  ........
*
```

The device index block, **statblk**, contains information on the devices configured into the system. The string "**ndev**" marks the word containing the size (number of entries) of **devblock**. Each class of device has an entry in the **statblk** that contains the following information:

- Character string used to identify the device type in a dump
- Base (the index into **statblk**)
- Count (the number of devices of the type that are configured)

The devices types include the following:

Device	Description
cpu	CPUs
disk	Disk drives
hsx	High-speed channels
hy	HYPERchannels
lsp	Low-speed channels
lsx	Full-duplex low-speed channels
stripe	Logical striped devices
ts	Tape subsystems

For example, the entry for the logical stripe device is marked with the string "**stripe**" at word 0107; word 0110 contains the value of **stripebase**, 0000000000000000000053; word 0111 contains the value of **nstripe**, 0000000000000000000001, which means that there is 1 logical stripe device configured.

Words 0060 through 0063 show the members of this logical stripe device. The following figure shows these words expanded as 8-bit bytes.

```
00000000060:  010 016  0   0   0   0   0   0
00000000061:    0   0  0   0   0   0   0   0
00000000062:    0   0  0   0   0   0   0   0
00000000063:    0   0  0   0   0   0   0   0
```

Layout of local memory

The kernel uses local memory words 0 through 0777. This area contains frequently used global variables and variables shared with user processes.

The file **md/lowlm.s** defines the position of these words in local memory.

Example: nmab(1) output

This example shows a portion of a dump of local memory from the command **nmab –nw** *kernel_binary*. The first column shows the absolute address, and the second column shows the symbol at that address.

```
/unicos - input file
numeric sort in local memory
    0   lowlmbottom
    0   cpunum
   01   sigdata
   02   sigoff
   03   sigpend
   04   cpelaps
   05   cpstart
   06   sigmask
   07   u
  010   uc
  011   up
  012   upc
  013   fcpudata
  014   kcpudata
  015   ostc
  016   semf
  017   lmtmctrl
  020   lmtmcrit
  021   lmtmint
  022   lmrt
  023   lmvm
  024   $fp
  024   $fpidnt
  025   $fprtrn
  026   $fpprev
  027   $fpbase
  030   $fptop
  031   $fpsegl
  033   $fpname
  033   fpname
  034   $lm
 0116   $scr
 0216   $args
 0242   usrregs
 0464   bssbss
 0464   lmend
 0465   INOUTsaveregs
 0605   ERROR$Oonereg
 0606   zzzzzzlm
```

Global pointers in local memory

The following global pointers are local memory resident:

Pointer	Description
u	Pointer to the **user** structure for the currently executing process
uc	Pointer to the user common (**ucomm**) structure for the currently executing process
up	Pointer to the process table (**proc**) entry for the currently executing process
upc	Pointer to the process common (**pcomm**) structure for the currently executing process

Throughout the kernel code, fields prefaced with **u->**, **uc->**, **up->**, and **upc->** are references to values in these structures.

Memory management restrictions

This subsection describes features of UNIX System V memory management that were not implemented on CRAY-2 systems. It then describes features of UNIX memory management that were implemented in UNICOS running on CRAY-2 systems.

Memory management features not on CRAY-2 systems

The CRAY-2 hardware has a simple base/limit method of memory management that makes the following UNIX features impossible to implement:

- Shared text
- Segmentation

- Paging
- Shared memory

When the UNIX System V operating system was ported to the CRAY-2 system, all code pertaining to shared text or memory was removed.

Memory management in UNICOS

On UNIX systems, the kernel addresses **user** structures by using a memory management trick; specifically, it uses a segment register or a page table to map each **user** structure to a constant address. The CRAY-2 hardware does not support use of segmentation or paging. Certain items have been moved from the **user** structure on a UNIX system to the **proc** structure in UNICOS, because UNICOS relies on these items staying at a fixed address.

The **user** structure is now referenced by a local memory pointer of the form **u->u_***xxx*, rather than a fixed address reference of the form **u.u_***xxx*.

The user process table pointer (**u.u_procp**) was moved into a global pointer (**up**) to eliminate a double pointer reference of the form **u->u_procp->p_***xxx*.

UNIX systems also expect the kernel stack to remain at a fixed address. UNICOS solves this problem by dereferencing the stack frame pointers when a stack is moved, and by not taking the address of any data resident on a stack.

File Management

UNICOS File Systems

This section describes the differences between file systems for UNIX System V and UNICOS running on CRAY-2 systems.

The design of the CRAY-2 native file system, C2FS, shows the influence of UNIX System V and of earlier COS systems. Like System V, UNICOS has a flat i-node space and a tree-structured directory that provide for file specification by i-node number (i-number) or by path name. Like COS, and unlike System V, larger files on UNICOS are allocated in larger (track-sized) units.

Differences between System V and UNICOS

The CRAY-2 file system contains the following differences from the file systems on UNIX System V:

- Bitmap-based track allocation for large files
- Multipartition file systems
- Flawing handled by user-level code
- Multiple i-node regions
- Super block split into two parts, the super block and the dynamic block
- Revised i-node format

The remainder of this subsection discusses these features.

Bitmap-based track allocation

In UNICOS, sector files are allocated from a free list, as they are in System V. However, large files use a bitmap-based track allocation method. This change was designed to improve I/O throughput.

In UNIX System V, sector-sized allocation is from the top of a push-down list of free sectors, which results in smaller allocations scattered throughout the file system; disk throughput is limited by the seek time needed to access each sector. The CRAY-2 system maintains the push-down free list for small files (files requiring 8 or fewer sectors). When a file requires the ninth sector, or if it is opened as a large file, the existing data is moved to the beginning of a disk track; further allocations are made in track-sized units. This speeds up I/O in two ways: (a) the allocations are in larger units, so more data may be moved in one I/O request; and (b) the allocator travels sequentially through the bitmap so that large files tend to be contiguous.

Track size (the number of sectors per track) depends on disk type. (See **disk**(4D) for specifications.) Note that copying large files between different types of disks changes a file's track usage.

The disadvantage of track allocation is that the space between the end of a file and the next track boundary is unused.

Multipartition file systems

In UNIX System V, a file system is limited to a portion of a physical device; this portion is called a *partition*. This limits the size of files to the size of the largest physical device. On a CRAY-2 system, however, several partitions can be grouped into a single file system. This provides a larger maximum file size and allows striping of files across several devices. This striping is called *background striping*. The disadvantage of striping is that the loss of an i-node on one partition may cause damage that extends beyond a single device.

UNICOS also supports *foreground striping* (also called *disk striping*). Foreground striping is used mainly for striping swap devices.

See "Disk Striping," page 129, for more information on foreground striping.

Flawing

In UNICOS, flawed sectors are handled by user-level code (**mkfs**(1M) and **fsck**(1M); note that **mkswp**(1M) is used to handle bad blocks for swap partitions). Together, these two programs keep flawed areas out of the free list. Consequently, kernel and driver software are not aware of the existence of flawed areas. The disadvantage to this method is that a raw disk backup procedure is more complicated in UNICOS than it is on systems that limit flawed sectors to the first part of a disk or those systems that use a redirect header to mark flawed sectors.

Multiple i-node regions

UNIX System V has a fixed number of i-nodes (fixed when **mkfs**(1M) is run) at the beginning of the file system. Since the i-nodes are often accessed by the kernel, this strategy causes excessive disk arm movement. In the CRAY-2 file system, **mkfs**(1M) establishes the i-nodes near the center of the partition. The provision for more than one i-node region permits the i-nodes to be mapped around flawed sectors and permits limited capability for expanding the number of i-nodes.

Super block and dynamic block

On UNICOS systems, the information contained in a UNIX super block is divided into a super block and a dynamic block. In UNIX System V, the super block is at the beginning of the disk;

it includes dynamic information such as the list of free sectors. However, since the dynamic information must be updated constantly to keep the file system reasonably consistent, and since the amount of information required in UNICOS overflows one sector, the UNICOS super block was divided.

The CRAY-2 super block contains static information that is needed only when the file system is mounted; this information includes, for example, the location of the i-nodes, the number of sectors, and the location of the overflow super block. This overflow super block is called the *dynamic block*; **mkfs**(1M) puts the dynamic block near the center of the partition. The information in the dynamic block includes the list of free sectors, the bitmap of free i-node areas, and the bitmap of free tracks. On some disk types, the bitmap of free tracks has grown to require a separate sector.

I-node format

In UNICOS, the format of an i-node has been modified to contain a bit indicating small or large format and 8 address pointers, instead of 13 in UNIX. For a small file, all of these pointers point to sector-sized blocks of data. Figure 2 shows the structure of an i-node for a small file.

For a large file (one consisting of more that 8 sectors), the first 5 pointers point to tracks of data. The sixth points to a sector containing 512 pointers to tracks of data. The seventh is double indirect, and the eighth is triple indirect.

In UNICOS, the maximum file size is as follows:

$$(5+512+512^2+512^3)*\mathit{track\_size}*4096$$

The value of *track_size* is 18, 42, or 48, depending on the disk drive type.

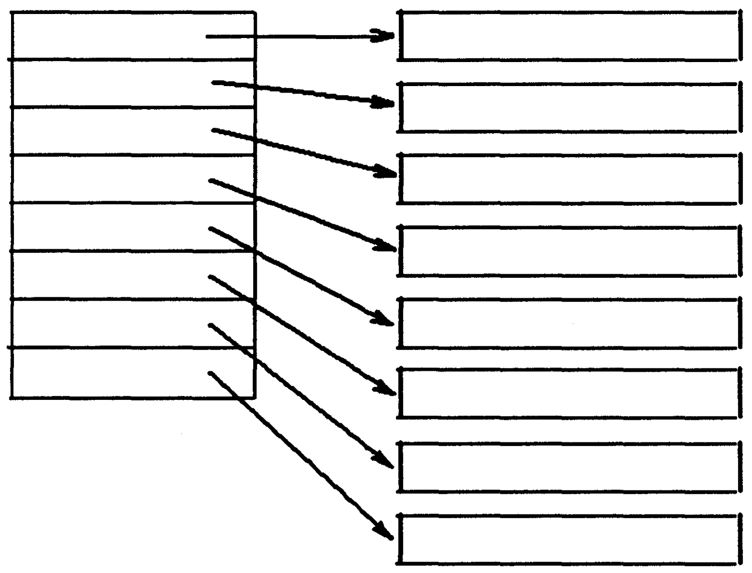

Figure 2. I-node format for a small file

File System Switch

The file system switch (FSS) provides a mechanism to implement more than one type of file system on a single machine. The file systems defined in release 5.0 of UNICOS are as follows:

- UNICOS native file system (C2FS)
- **/proc** file system (PROC)
- UNICOS Network File System (NFS)
- SUPERLINK file system (SLFS)

This section describes the kernel-level implementation of the FSS. It covers the FSS mechanism, the flow of control, the FSS data structures, and the FSS routines.

Overview of the file system switch

In some versions of UNIX, the system call layer interfaces directly with the file system routines; in UNICOS, the FSS layer contains the file-system-independent routines that access specific file systems. Figure 3 displays the relationship of the FSS to the rest of the system.

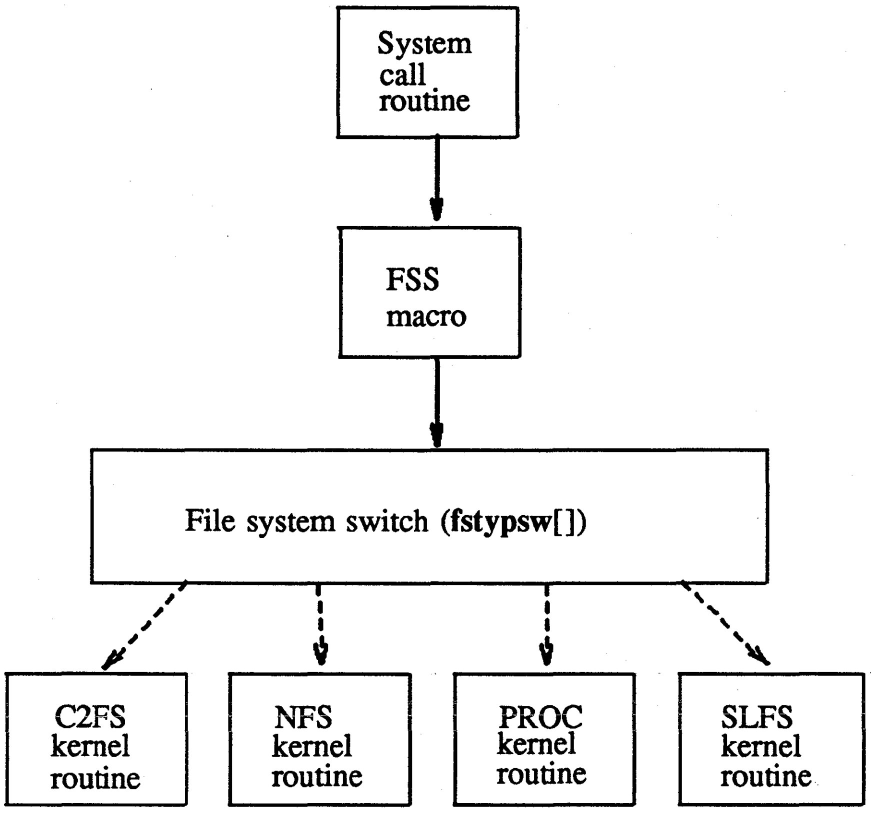

Figure 3. Overview of the file system switch

The actual mechanism for the FSS consists of an array of file-system-specific routines; this array is indexed by the file system type. File-system-independent routines call the file-system-specific routines through this array, which is called **fstypsw**. The **fstypsw** array is defined in the include file **sys/conf.h**, and is initialized in the file **cf/conf.c**.

Each i-node contains a pointer to this array in **i_fstypp**. The FSS macros (defined in **sys/fstyp.h**) use this pointer to access the correct file-system-specific routine for each operation.

The following example of the FSS uses the **open**(2) system call and the C2FS file system. The system performs the following actions to access the file-system-specific **open** routine in the kernel.

1. Calls the C library **open**() routine (**lib/libc/sys/open.s**).
2. Enters the system through the **sysent** table (defined in **os/sysent.c**). (See "System Calls," page 65, for more information on the **sysent** table.)
3. Calls the kernel-level routine **open**() (defined in **os/sys2.c**).
4. Calls the routine **copen**() (defined in **os/sys2.c**). **copen**() performs all operations needed to verify access to the file.
5. Calls **cfinopen**() (defined in **os/sys2.c**). **cfinopen**() completes the open operation.
6. Uses the macro **FS_OPENI** (defined in **sys/fstyp.h**) to access the file-system-specific **openi** routine.
7. Calls the **c2openi**() routine (defined in **fs/c2/c2subr.c**); this routine actually opens the file on a C2FS file system.

FSS data structures

The data structures, constants, and variables in the FSS code are as follows:

Item	Description
fstypsw	The FSS table; it is defined in **sys/conf.h**. This structure defines all entry points to each file system.
fsinfo	This structure contains general and configuration information about each file system. It is defined in the include file **sys/conf.h**.
i_fstypp	Pointer in the i-node into the FSS table (**fstypsw**) for the appropriate file system.
inode	This structure defines a generic i-node; each active file has an **inode** structure allocated to it. Fields in this structure are used and assigned by each file system type. The **inode** structure contains a pointer to a file-system-specific area used only by the associated file system.

Item	Description
mount	This structure defines the mount point for a file system; each mounted file system has a **mount** structure allocated to it.

File-system-specific routines

The kernel uses the following *fstyp names* (defined in the include file **sys/fsid.h**) to map the file system type to an index in the **fstypsw** table.

```
#define C2FS    "C2FS"    /* CRAY-2 file system          */
#define NFS     "NFS"     /* NFS file system             */
#define PROC    "PROC"    /* /proc debugger file system  */
#define SLFS    "SLFS"    /* SUPERLINK file system       */
```

These names are used for the **sysfs**(2) system call.

The symbols **SLON** and **CRINFS** are used to enable or disable code that is applicable only if the specific file system types are built into the kernel. UNICOS requires the C2FS and **/proc** file systems to boot.

Native file system routines

The file-system-specific routines for the native system, C2FS, are found in the directory **uts/fs/c2**.

UNICOS NFS file system routines

The file-system-specific routines for the UNICOS NFS file system are found in the directories **/usr/src/net/nfs/fs/nfs** (the source files) and **/usr/src/net/rpc/include/sys/fs/nfs** (the include files). These routines are included in the FSS table, **fstypsw**, if the symbol **CRINFS** is defined.

See the *UNICOS NFS Internal Reference Manual*, publication SM-2065, for more information about the UNICOS NFS routines.

/proc file system routines

The file-system-specific routines for the **/proc** file system are found in the directory **uts/fs/proc.**

See the *UNICOS System Administrator's Guide for CRAY-2 Computer Systems*, publication SG-2019, for more information about the **/proc** file system.

SUPERLINK file system routines

The file-system-specific routines for the SUPERLINK file system are found in the directory **uts/fs/sl.** These routines are included in the FSS table, **fstypsw,** if the symbol **SLON** is defined.

See the *SUPERLINK UNICOS Installation, Tuning, and Customization Guide*, publication SI-0187, for more information on SLFS.

Data Migration

The UNICOS data migration facility manages the on-line mass storage systems on a Cray computer system. It allows you to keep a specified amount of disk space available on a UNICOS file system by migrating files off-line to magnetic tape.

A number of changes were made to the UNICOS kernel for data migration. These changes can be summarized as follows:

- The UNICOS file system structure has been changed to support data migration.
- The UNICOS i-node structure has been changed to support data migration.
- The **dmmode**(2) system call has been added to allow user selection of automatic file retrieval.
- The **dmofrq**(2) system call has been added to allow user access to information on the status of migrated files and to implement data migration requests from the data migration daemon (**dmdaemon**(1M)) to the kernel.
- A data migration driver has been added to provide the kernel interface.

This section describes the kernel-level portion of the data migration feature.

Enabling the data migration feature

The kernel makefile uses the symbol **DM_SYS** to control the inclusion of many portions of the data migration feature.

When **DM_SYS** is set to 1 in the kernel makefile, the UNICOS kernel is built with all of the data migration features enabled. If the kernel is built with **DM_SYS** undefined (or set to 0), the off-line files are recognized, but cannot be recalled, and new files cannot be migrated.

User interface to data migration

The data migration facility allows the kernel to initiate retrieval of an off-line file. In addition, each process has a flag (**uc_dmode**) in the user common (**ucomm**) area that allows the user to determine which processes can use automatic retrieval. The flag can be set with the **dmmode**(1) command and the **dmmode**(2) system call. A value of 1 indicates that the file can be retrieved (automatic retrieval is enabled); a value of 0 indicates that the file cannot be opened (automatic retrieval is disabled). The flag is inherited as part of the environment for a child process. The default value for this flag is defined by **DMODE** in the include file **sys/param.h**. However, regardless of the value, any user can use the **dmget**(1) command to request the recall of a migrated file.

When a process attempts to open a migrated file, there are two possible results:

- The process pauses, waiting for the file to be staged back to Cray disks.
- The process fails to open the file with a system error **EOFFLIN** (file off-line, no automatic retrieval).

File system changes

A significant portion of the changes necessary for data migration were implemented in the kernel to avoid changing existing file systems. Additionally, file systems are upward-compatible; that is, a UNICOS 4.0 file system can be used with UNICOS 5.0. When files are migrated (that is, moved off-line) under UNICOS

5.0, those files are no longer accessible under UNICOS 4.0; unless compatibility mods are added to UNICOS 4.0, the 4.0 version of **fsck**(1M) removes the i-nodes for migrated files. However, if all off-line files are retrieved under UNICOS 5.0, the file system runs under UNICOS 4.0 with no undesired effects.

Two file types, **IFOFL** and **IFOFD**, have been added to the UNICOS kernel. Like a regular file, a migrated file has an i-node. The type of a migrated file is identified in the i-node's mode field as **IFOFL** (a regular file is identified by **IFREG**). The **IFOFD** file type is a transition type used when data blocks from a file are being copied to another i-node. This transition state should exist only for an instant, but has been defined to prevent corruption of the file system, or the file's data blocks, when a file is being moved.

Files of type **IFOFL** appear to reside in the UNICOS file system; however, these files are off-line and have no data blocks associated with them. The i-node for an **IFOFL** file contains a *file handle*, which is a pointer to the off-line data for the file. This file handle is 2 words long and is written into the data block area of the disk i-node. **dmdaemon**(1M) assigns the file handle.

A migrated file appears as a regular file (**IFREG**) to most UNICOS utilities. The **stat**(2) and **fstat**(2) system calls have been modified to return **IFREG** instead of **IFOFD** or **IFOFL** for an off-line file. The **dmofrq**(2) system call can be used to determine whether an off-line file is of type **IFOFD** or **IFOFL**.

Data migration system calls

The user interface to data migration consists of two system calls: **dmmode**(2) and **dmofrq**(2). The **dmmode**(2) system call selects (toggles) automatic file retrieval, and the **dmofrq**(2) system call returns information on the status of migrated files and processes requests from the data migration daemon.

The **dmmode**(2) system call implements user selection of automatic retrieval. As described in "User interface to data migration," page 60, **dmmode**(2) can set or clear the **uc_dmode** flag to toggle automatic retrieval of migrated files.

The **dmofrq**(2) system call processes requests from **dmdaemon**(1M); it can also be used to return information on the status of migrated files. The following functions are available:

- Indicate that a file is to be changed from **IFREG** to **IFOFL** with the specified handle information. (Used when the premigration entry creation has been completed; data blocks are not returned to the file system, but the data block area of the specified file's i-node is cleared.)
- Specify reading and writing the file handle from or to the off line file i-node.
- Transform an **IFOFD** file to an **IFREG** file. The original i-node is transformed when being retrieved; the premigration copy is transformed when being migrated.
- Move the data block area in an **IFREG** i-node.
- Return a **stat**() block with the mode unmodified.

The data migration driver

The data migration driver (see **dmd**(4D)) is the interface between the kernel and **dmdaemon**(1M). The driver's function is to recall, restore, and remove off-line files, as requested by **dmdaemon**(1M). The driver is defined in the file **io/mig.c**.

The parameter **KERNEL_DAEMON** (in **sys/dmkreq.h**) defines the special file (device node) used for the device driver. By default, **KERNEL_DAEMON** is **/dev/dm/mig0**. The include file **sys/dmkreq.h** also contains the definitions for the **ioctl**(2) requests for the driver.

The driver typically passes messages from the kernel to **dmdaemon**(1M) to recall and remove off-line files. Only one user (usually **dmdaemon**(1M)) is allowed to have the device (**/dev/dm/mig0**) open. (Code in the open routines ensures this; in practice, only the daemon should open this special file.)

The recall message (**KDMRQ_RCL**) is used to recall an off-line file. When an off-line file is being opened, the kernel routine **copeni**() calls **dmopeni**() (in module **os/dmsys.c**). **dmopeni**() sends the recall message, which is queued awaiting a reply message from **dmdaemon**(1M). The open operation waits until the file has been retrieved (that is, the process sleeps at an interruptible priority). If the process is interrupted (by the interrupt signal), the error **EOFLRIN** is returned.

A remove message (**KDMRQ_VDC**) is used to remove an off-line file. It is sent when the user removes an off-line file, thus voiding any off-line copies made. The processing for a remove message is similar to that for a recall message, but the process does not wait for a reply from the daemon.

Process Management

System Calls

System calls provide the mechanism for user processes to communicate with and make requests of the UNICOS kernel. This section describes the interface between the user-level routines, C library routines invoked in user programs, and the kernel-level routines that perform the work in the kernel.

The user-level system call routines are described in detail in *Volume 4: UNICOS System Calls Reference Manual*, publication SR-2012. This manual does not attempt to duplicate that documentation; rather, it describes the lower levels of the system call interface to the kernel. Specifically, this subsection addresses the following topics:

- Categories of system calls
- Origin of UNICOS system calls
- System call interface
- Flow of control during system call execution

This section also provides an example of the execution of a simple system call (**umask**(2)).

Categories of system calls

There are three categories of system calls:

- *True system calls*; the library interfaces for these system calls transfer control directly to the kernel routine that performs the system call. Most system calls fall into this category. An example of a true system call is **getpid**(2).

- *Preprocessed system calls*; the library interfaces for these system calls perform some preprocessing before transferring control to the kernel. The **exec** family of system calls (for example, **execl**(2)) includes system calls that are preprocessed.
- *Pseudo system calls*; the library interfaces for these system calls perform all processing at the library level. In other words, control is never transferred to the kernel with an **exit** instruction. There are few system calls in this category. An example of a pseudo system call is the **sigblock**(2) system call, which manipulates a word in local memory; it does not need to transfer control to the kernel for any processing.

UNIX System V system calls

The following UNICOS system calls are derived from AT&T UNIX System V release 3:

fstatfs(2)
getdents(2)
mkdir(2)
rmdir(2)
sighold(2)
sigignore(2)
sigpause(2)
sigrelse(2)
sigset(2)
statfs(2)
sysfs(2)

The following UNICOS system calls are derived from versions of UNIX System V prior to release 3:

access(2)
acct(2)
alarm(2)
brk(2)
chdir(2)
chmod(2)
chown(2)
chroot(2)
close(2)
creat(2)
dup(2)
exec(2)
exece(2)
exit(2)
fcntl(2)
fork(2)
fstat(2)
getgid(2)
getpid(2)
getuid(2)
ioctl(2)
kill(2)
link(2)
lseek(2)

mknod(2)
mount(2)
nice(2)
open(2)
pause(2)
pipe(2)
plock(2)
profil(2)
ptrace(2)
read(2)
setgid(2)
setpgrp(2)
setuid(2)
stat(2)
stime(2)
sync(2)
time(2)
times(2)
ulimit(2)
umask(2)
umount(2)
uname(2)
unlink(2)
ustat(2)
utime(2)
utimes(2)
wait(2)
write(2)

UNIX 4.2BSD system calls

The following UNICOS system calls are derived from UNIX 4.2BSD.

getgroups(2)
gettimeofday(2)
setgroups(2)
settimeofday(2)
sigblock(2)
sigsetmask(2)

UNICOS system calls

Table 2 lists the system calls available only on Cray computer systems running UNICOS.

Table 2. UNICOS system calls

System call	Description
acctid(2)	Changes account ID of a process
chacid(2)	Changes disk file account ID
chkpnt(2)	Checkpoints a process, multitask group, or job
cpselect(2)	Selects which processors may run the process
dmmode(2)	Sets and gets data management retrieval mode
dmofrq(2)	Processes off-line file requests
fsecstat(2)	Gets file security status (available on secure systems only)
getfacl(2)	Gets access control list for a file (available on secure systems only)
getfcmp(2)	Gets compartments of named file (available on secure systems only)
getflvl(2)	Gets security level of named file (available on secure systems only)
getjtab(2)	Gets the job table entry associated with a process
getsysl(2)	Gets minimum and maximum security levels for the system (available on secure systems only)
getucmp(2)	Gets user's active compartments (available on secure systems only)
getulvl(2)	Gets user's active security level (available on secure systems only)
getusrv(2)	Gets user's security validation information (available on secure systems only)
ialloc(2)	Allocates storage for a file
jacct(2)	Enables or disables job accounting
killm(2)	Sends a signal to a group of processes
limit(2)	Sets resource limits
limits(2)	Returns or sets limits structure
listio(2)	Initiates a list of I/O requests
mtimes(2)	Multitasking execution overlap profile
nicem(2)	Changes priority of processes or group of processes
reada(2)	Asynchronous read from file
resch(2)	Reschedules a process
resume(2)	Resumes execution of processes

System call	Description
restart(2)	Restarts a process, multitask group, or job
rmfacl(2)	Removes an access control list from a file (available on secure systems only)
secstat(2)	Gets file security status (available on secure systems only)
select(2)	Examines synchronous I/O multiplexing
setfacl(2)	Sets access control list for a file (available on secure systems only)
setfcmp(2)	Sets file's security compartments (available on secure systems only)
setflvl(2)	Sets directory or file security level (available on secure systems only)
setjob(2)	Sets job ID
setsysl(2)	Sets security levels for the system (available on secure systems only)
settfm(2)	Sets and gets trusted facility mask (available on secure systems only)
setucmp(2)	Sets user's active security compartments (available on secure systems only)
setulvl(2)	Sets user's active security level (available on secure systems only)
setusrv(2)	Sets user's validation information (available on secure systems only)
sigctl(2)	Provides generalized signal control
slgentry(2)	Makes security log entry (available on secure systems only)
suspend(2)	Suspends execution of processes
sysconf(2)	Retrieves system implementation information
target(2)	Retrieves or modifies machine characteristics
tfork(2)	Creates a multitasking process
trunc(2)	Truncates a file
upanic(2)	User panic; stops the system from a user process
waitjob(2)	Gets information about a terminated child job
writea(2)	Performs asynchronous write on a file

System call interface

This subsection describes the C library routines involved with system calls, the **sysent** table, and the kernel-level system call routines.

C library routines

System calls are generally invoked through a C library interface. This is a predefined library of functions (the C library) whose names correspond or are similar to the system call names. The source for these interface routines is in **lib/libc/sys**. These routines use the **exit 1** instruction (usually by using the **SYSCALL** macro, defined in **lib/csu/asdef.s**), which changes the process execution from user mode to kernel mode and causes the kernel to start executing code for system calls.

The file **lib/csu/asdef.s** contains the C library's definitions of the system call numbers.

The sysent table

The file **uts/os/sysent.c** contains the **sysent** table. This is the system call entry point table; it contains information for each system call.

The kernel uses the system call number as an index into this table to find the entry point for the system call and to find the number of parameters that the system call expects.

The format of an entry in the **sysent** table is defined in the include file **sys/sysent.h**.

Flow of control

Figure 4 shows the flow of control in system call handling.

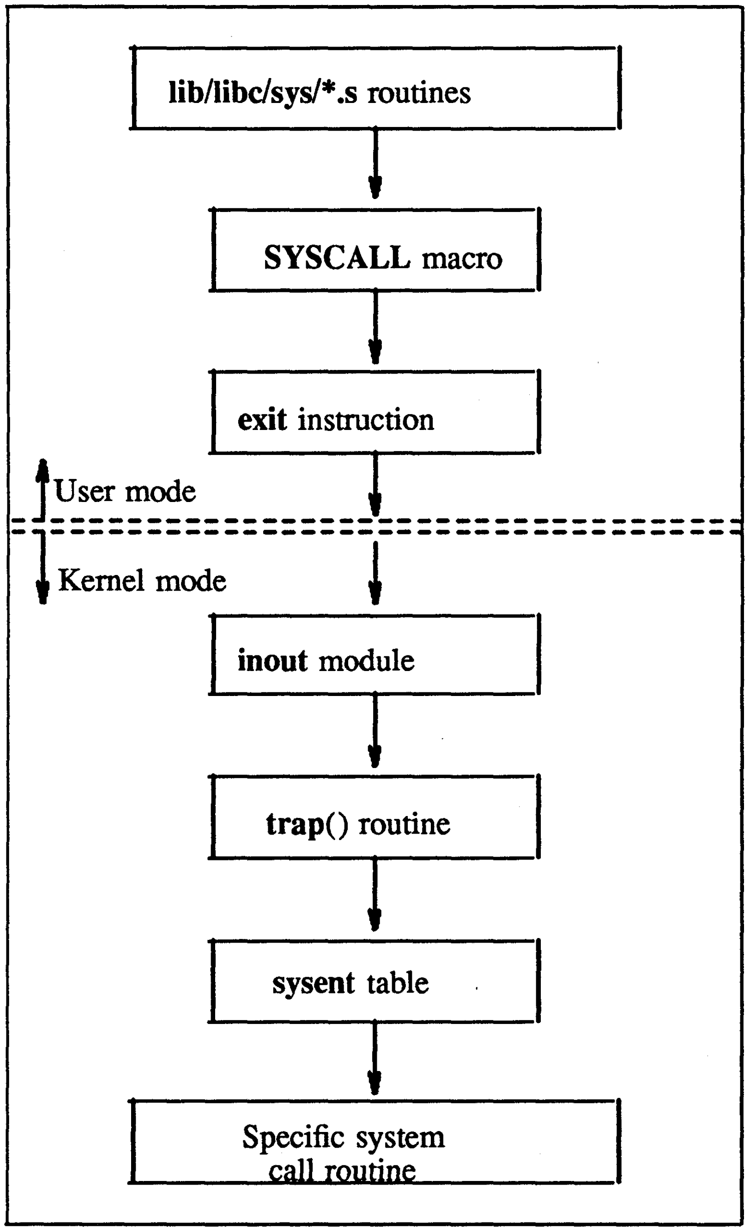

Figure 4. System call flow of control

Example: Flow of control in umask(2) system call

The following example shows the flow of control for the **umask**(2) system call. This system call is used as an example because it is a true system call; no library level processing is done.

The flow of control in a call to **umask** is as follows:

1. A user program calls **umask** as in the following example:

   newmask = umask(*cmask*);

   The C language calling sequence copies the contents of *cmask* into local memory and calls the **umask** function (**lib/libc/sys/umask.s**).

2. The **$SYSCALL** macro places a pointer to the local memory arguments in register A1 and the system call number (defined in **lib/asdef/asdef.s**) in register S0. The upper eight bits of S0 are set to 1 for most system calls; this indicates that the vector registers do not need to be saved.

3. An **EXIT 1** instruction is executed. The background processor (BP) stops execution, sets the idle and exit bits in the status word and waits for the foreground processor (FP) to sense the exit.

4. The FP saves the P and S registers of the BP, switches it to kernel mode and begins execution with the P/S/BA/LA in the operating system inbound exchange package. Execution begins at **p***NN***gosys** (*NN* is the CPU number the system call is executing in). The entry and exit code for the kernel is called *the EXEC*; this code is located in **uts/md/inout.s**.

5. The kernel saves the user's registers, initializes several of its variables, updates user/system time, acquires the *kernel lock* (a common memory lock), then calls the **trap**() function (**uts/md/trap.c**).

6. **trap**() copies the system call arguments pointed to by A1 from local memory to an array in the **proc** structure. The three system call return values (**RVAL0**, **RVAL1** and **RVAL2**, which correspond to the user's saved S0, S1, and S2, respectively) are cleared. If the system call is interruptible (for example, **open**(2), **read**(2), or **write**(2)), **trap**() calls **setjmp** to set up a return address for the interrupt. Finally, the system call number in S0 is used as an index in the **sysent** table and the proper routine is called. In this example, S0 equals 60; the corresponding **sysent** table is as follows:

```
SYSENT( "umask",     1,  0, umask)     /* : umask */
```

7. The **umask**() function (**uts/os/sys4.c**) is called; the previous mask is stored in **RVAL1** and the user's mask (**uc->uc_cmask**) is set from the value passed in by the user. The **umask**() function is as follows:

```
/*
 * mode mask for creation of files
 */
umask()
{
        register struct a {
                int     mask;
        } *uap = (struct a *)up->p_arg;
        register int t;

        t = uc->uc_cmask;
        uc->uc_cmask = uap->mask & 0777;
        RVAL1 = t;
}
```

8. The **umask**() function returns to **trap**(). If there was an error during the system call execution, **u->u_error** is copied to **RVAL0**. **trap**() updates some scheduling information and checks the **runrun** and **runwout** variables to see if this process should be rescheduled or should process wayout entries.

9. **trap**() returns to the EXEC. The **devblock** table is scanned for interrupts arriving during the system call execution, the user's registers are restored, and the kernel executes an **EXIT 1** instruction to notify the FP to return this CPU to user mode.

10. The FP loads the user's saved P/S/BA/LA into the BP and resumes execution at the instruction following the user's **EXIT 1**.

11. If S0 is set, this error number is copied into the user's global variable **errno** and S1 is set to –1. If S0 is not set, the correct return value is already in register S1, so the system call simply returns to its caller.

12. The calling sequence copies the value in S1 to the user's variable *newmask*.

Signals

This section discusses UNICOS signals. It covers the following topics:

- Signal differences between UNICOS and UNIX System V
- Signal kernel routines
- Signal macros
- Data areas
- Signal differences between CRAY-2 systems and CX/1 and CEA systems

Differences of UNICOS signals

The differences between UNICOS signals and signals in UNIX System V are summarized as follows:

- In UNICOS, more signal management is done at the library level than at the kernel level.
- More signals (64) are available in UNICOS.
- The UNICOS signal routines are a mixture of routines from UNIX System V release 2, UNIX System V release 3, UNIX 4.2BSD and routines unique to Cray systems. The kernel signal interface is through the **sigctl**(2) system call; this system call provides more general control than is available on UNIX systems.

- User processes have control of the default action of a signal.
- Some UNICOS signal names differ from UNIX signal names (for example, in UNICOS, **SIGABRT** is used instead of **SIGIOT**).

Signal kernel routines

The following routines are involved in the low-level management of signal processing.

Routine	Description
fsig()	This routine gets the lowest-order bit in **p_sig**, the mask of signals that have been sent to the process. **fsig**() is in the module **md/fsig.s**.
issig()	This routine tests for the existence of a signal. **issig**() is called for each process as it enters the kernel. If there are signals pending that are not ignored, **issig**() returns true (unless the signal is **SIGCLD** or **SIGPWR**). If the signal is **SIGCLD** or **SIGPWR**, UNICOS processes them in the same way that UNIX does. **issig**() is in the module **os/sig.c**.
psig()	This routine is called if **issig**() returns with a signal to process. **psig**() determines what will happen when a signal is seen by a process; the process receiving the signal executes **psig**() to perform one of three actions: • Send the signal if the process has registered for it. **psig**() sends a process a signal by calling **sendsig**() and then clears the **p_sig** bit associated with the signal. • Dump process memory (do a core dump) and die. • Die without doing a core dump. **psig**() is in the module **os/sig.c**.

Routine	Description
psignal()	This routine sends a signal. Unless the receiving process is the swapper (process 0) or an idle process, **psignal**() sets the signal bit in **p_sig**. If the process is being killed with the **SIGKILL** signal and is currently suspended, the suspend is cleared so that the process can be killed. Processes sleeping at interruptible priorities are awakened, and processes connected in another CPU are interrupted for quick servicing. This is the only signal routine ever executed by a process other than the one receiving the signal. **psignal**() is in the module **os/sig.c**.
retsig()	This routine is called when the user makes a **sigctl**(2) call with the **SCTL_RET** action. The library does not restore all the registers (P, S0, S1, S2, and A1 registers), so **retsig**() must restore them. In addition, **retsig**() sets the semaphore pointer in case it was changed by the kernel. **retsig**() is in the module **os/sig.c**.
sendsig()	This routine is called by **psig**() to send a signal to the user process. **sendsig**() checks the signal nesting level (if greater than 10, **sendsig**() returns to try again later). It then checks the top stack entry; if the entry can be reused, **sendsig**() does so; otherwise, it gets another stack entry. **sendsig**() is in the module **os/sig.c**.
sigctl()	Entry point (in **os/sys4.c**) for the **sigctl** family of routines.
signal()	This routine (in **os/sys4.c**) sends the specified signal to all processes with **pgrp** as their process group. It calls **psignal**() to actually send the signal.

Macros

The following macros are used in signal processing:

Macro	Description
SIGOFF	Macro to turn signals off for the currently running process. This macro is defined in the file **lib/csu/asdef.s.**
SIGON	Macro to turn signals on for the currently running process. This macro is defined in the file **lib/csu/asdef.s.**

Data areas

The **user** structure contains a storage area, the **u_sigenv**[] array, used to save the user context during signal processing. The process (**proc**) structure contains the field **p_sig**, which is a bitmask of signals pending for the process.

Several words in local memory are used in signal processing. These words are described as follows:

Memory Word	Description
_sigdata	Used to hold an incoming signal. The **_sigdata** word has the following format:
	| **os_flag** | *signo* | *P-address* |
	The first field, **os_flag**, is 24 bits; it is used in only one routine to check the version of signal handling. The *signo* field is 8 bits; it holds the signal number. The *P-address* field is 32 bits; it holds the address of the user process (the point at which it was interrupted).
_sigoff	Used to hold the sigoff word; this word is set to 0 to enable signal trapping and to a nonzero value to disable signal trapping.

Memory Word	Description
_sigpend	Used to hold signals that come in when signal trapping has been disabled.
_sigmask	Used to hold the signal mask.
_sigsave	Used to store the **_sigoff** word when signals are disabled.

Signal differences between CRAY-2 and CX/1 and CEA systems

There are several differences between the implementation of signals on CRAY-2 computer systems and the implementation of signals on CX/1 and CEA computer systems.

Most of the differences are due to architectural or hardware differences; for example, signals on CRAY-2 systems use local memory, which is not available on the other systems. However, there are also some software differences. (Note, however, that software differences might not remain in future releases of UNICOS.) These differences are as follows:

- CX/1 and CEA systems allow unlimited signal nesting; CRAY-2 systems limit nesting to **SIGNEST** levels. (**SIGNEST** is defined in **sys/signal.h**.)
- On CX/1 and CEA systems, all signal functions result in system calls, except **sigon**(3C) and **sigoff**(3C), which are library routines. On CRAY-2 systems, most signal functions are implemented at the library level and utilize local memory.
- CX/1 and CEA systems define but do not use the **SIGBUFIO**, **SIGMT**, and **SIGMTKILL** signals. These signals are used for Fortran and multitasking programs on CRAY-2 systems.
- CRAY-2 systems define but do not use the **SIGUME** and **SIGDLK** (**SIGCRAY7**) signals. On CX/1 and CEA systems, these signals are used upon encountering uncorrectable memory errors (UMEs) and hardware deadlocks, respectively.

Interrupts

Typical UNIX machines have a multilevel priority system whereby activities with higher priority can preempt those with lower priority. The UNIX structure assumes a multilevel environment, and low-priority activities proceed to completion without checking to see if something else should be done first. Typically, disk interrupts would preempt terminal interrupt servicing, and terminal interrupts would preempt system call servicing.

The CRAY-2 architecture is only a two-level system in which user mode is always interruptible and kernel mode never is. Two UNICOS files, **md/inout.s** and **md/trap.c**, are used in interrupt processing. **md/inout.s** contains a CAL routine that contains the **intscan** entry point and the **INTSCAN** macro, which scans for interrupts.

The remainder of this section describes the **ENABLE** and **DISABLE** macros and the UNICOS interrupt handlers.

The ENABLE and DISABLE macros

To prevent interrupts at inopportune times, UNIX artificially raises its priority with an **spl** instruction and then lowers it at the end of the critical region with an **spl0**. UNICOS uses the **ENABLE** and **DISABLE** macros to perform the same function.

These macros are defined in the include file **sys/sysmacros.h**; they have the following functions:

Macro	Description
ENABLE	This macro lets interrupts occur (reach the process).
DISABLE	This macro keeps interrupts from reaching the process. If an interrupt is in progress when this macro is invoked, the interrupt processing is finished before the disable action takes effect.
xENABLE	This macro is the nested version of the **ENABLE** macro.
xDISABLE	This macro is the nested version of the **DISABLE** macro.

The interrupt handlers

The device control block, **devblock** (defined in **sys/devdata.h**), contains the interrupt handler addresses for the devices configured on the system. (Interrupt handlers are also called *device interrupt service routines.*)

For more information on interrupt handlers and **devblock**, see "Kernel Data Structures," page 15, and "Foreground Processor and Kernel Communication," page 33.

Process Management Chains

This section describes the extensions to the process table for process management. These extensions consist of linked lists, termed *process management chains*, in the process table entry; they are used to decrease search time for entries.

In previous versions of UNICOS and in the UNIX operating system, the process table consisted of entries that were ordered sequentially, as they were created. UNICOS now has several process management chains that provide access to the following elements:

- The next available entry in the process table
- All active processes, starting with the youngest (most recently created) process
- All processes on a particular hash chain
- The most recent child of a particular process
- All the sibling processes for a process
- All the processes in a multitask group, starting with the oldest (first) process
- All the processes in a job

These chains reduce the need to perform linear search operations to find an entry in the process table. Consequently, several system calls (for example, **kill**(2), **killm**(2), **nicem**(2), and **wait**(2)) run considerably faster. In addition, the scan operations in **clock**(), **swtch**(), and **sched**()) are more efficient.

The **chkpnt**(2) and **restart**(2) system calls depend heavily on the existence of several of these extensions (for example, the chains listing the **pcomm** areas in a process group).

The fields of interest in the process structure (**proc**) are as follows:

struct proc *p_pidhash	Next process in the process ID (PID) hash chain
struct pcomm *p_children	Most recent child process
struct proc *p_prev	Previous process in the chain
struct proc *p_next	Next process in the chain

See the include file **sys/proc.h** for more information on the **proc** structure.

The fields of interest in the **pcomm** area are as follows:

struct proc *pc_mproc	Pointer to oldest process in a multitask group
struct pcomm *pc_pgrpprev	Pointer to previous process in the same process group
struct pcomm *pc_pgrpnext	Pointer to next process in the same process group
struct pcomm *pc_sibling	Pointer to next sibling in a multitask group
struct pcomm *pc_jlink	Pointer to next process in a job

See the include file **sys/proc.h** for more information on the **pcomm** structure.

The remainder of this section describes these chains.

Chains in the proc structure

The process management extensions added three chains to the process table entry structure, **proc**. These chains are as follows:

- Process ID (PID) hash chains
- Active process chain
- Available process chain

The following subsections describe these chains. In addition, the **proc** structure contains a child process pointer; this pointer is also described.

PID hash chains

The PID hash chains provide a method of mapping a PID to a process table entry. The PID hash table, **pidhash**[], is an array of pointers, one for each hash chain; each process is stored at an index determined by its hash value. Each entry in the **pidhash**[] array is the head of a singly-linked list of **proc** structures. Each hash chain holds all the process table entries for the processes whose PIDs hash to the same value.

The field **p_pidhash** points to the next entry on a hash chain. Each hash chain is NULL terminated.

There are **NPIDHASH** hash chains in the PID hash table. **NPIDHASH** is defined in the include file **sys/param.h**.

The macro **PIDHASH**() provides a hash function for the process table entries. It returns an index into the PID hash table. (This value is in the range $0 \leq \textit{value} <$ **NPIDHASH**.) **PIDHASH**() is defined in the include file **sys/sysmacros.h**.

Figure 5 shows a simplified view of the PID hash chain.

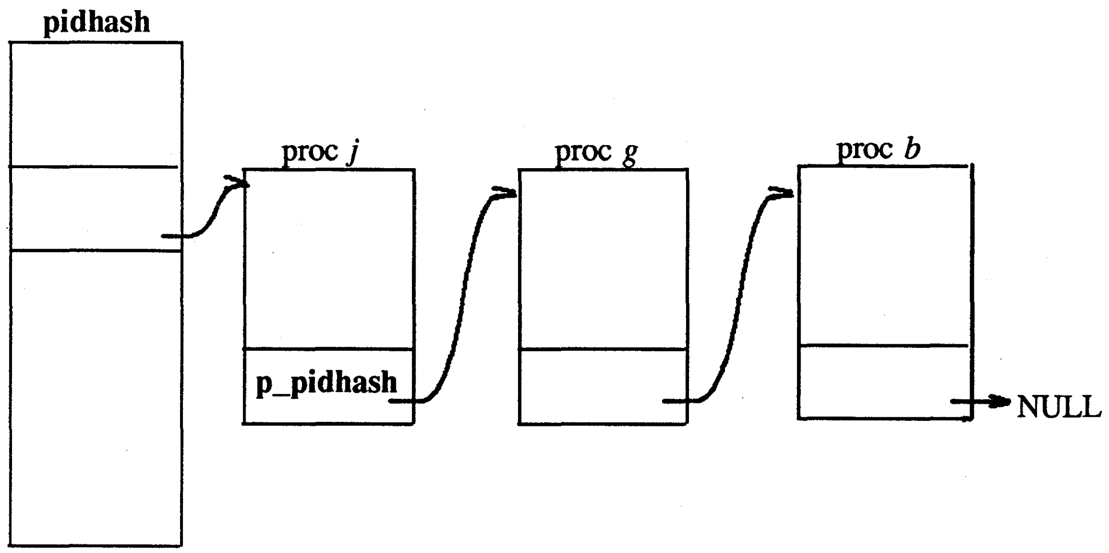

Figure 5. PID hash chain

Active process chain

The active process chain is a doubly-linked list of active (allocated) process table entries. In addition to user processes, this chain includes **sched**() (**proc[0]**, or the swapper), the idle processes, **init**, and any uncollected zombie processes.

The pointer **allproc** points to the most recently created process in the system. **p_next** is used as a pointer to the next older active process. **p_prev** is used as a pointer to the next younger active process.

Figure 6 shows a simplified view of the active process chain.

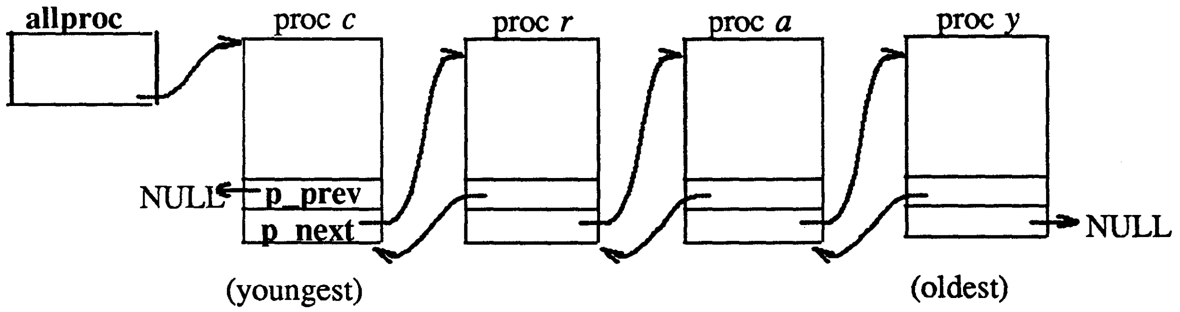

Figure 6. Active process chain

Available process chain

The available process chain is a singly-linked list of available (free) process table entries. The structure **availproc** is a pointer to the first entry in the chain. **p_next** is used as a pointer to the next available process. This chain is NULL terminated.

Figure 7 shows a simplified view of the available process chain.

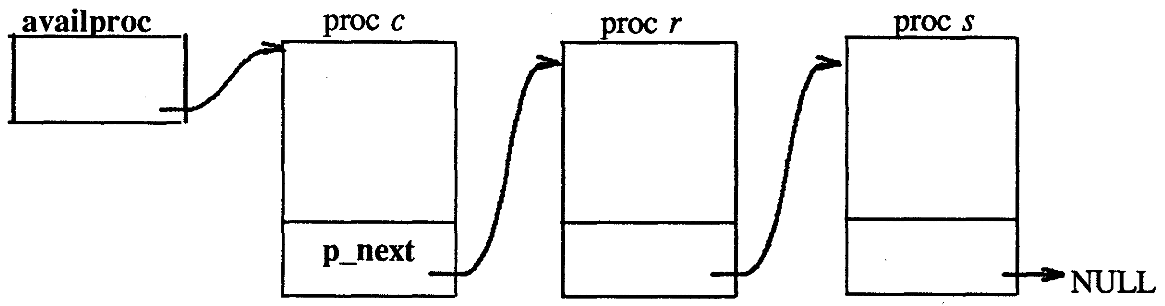

Figure 7. Available process chain

Child process pointer

The **p_children** field is used as a pointer to the most recent child of a given process.

To find all the children of a process *n*, follow process *n*'s **p_children** pointer to find its most recent child process, then use the **pc_sibling** field in that child process' **pcomm** area to find all of process *n*'s earlier children. (The **pc_sibling** field is described in "Sibling process chain," page 90.)

Figure 8 shows a simplified view of the child process pointer.

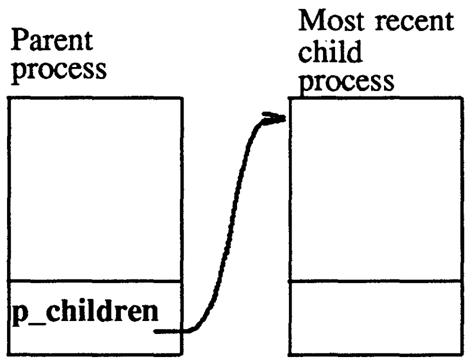

Figure 8. Child process pointer

Chains in the pcomm structure

The process management extensions added three chains to the process common area, **pcomm**. These chains are as follows:

- Sibling process chain
- Chain of processes in a process group
- Chain of processes in a job

The following subsections describe these chains. In addition, the **pcomm** structure contains a pointer to processes in a multitask group; this pointer is also described.

Sibling process chain

The sibling process chain is a singly-linked list of sibling processes in a process group. The field **pc_sibling** is a pointer to the **pcomm** area of another sibling process in the same process group. This chain is NULL terminated.

Figure 9 shows a simplified view of the sibling process chain.

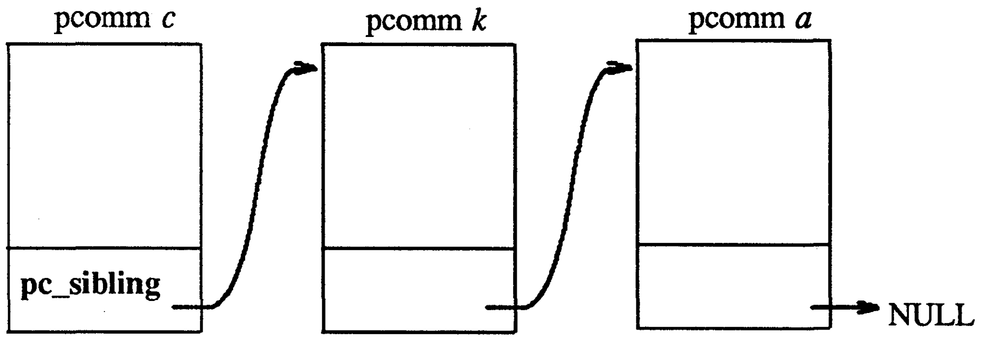

Figure 9. Sibling process chain

Chain of processes in a process group

This chain is a doubly-linked list of all processes (**pcomm** areas) in a process group. (The process group leader is indicated by the field **pc_pgrp**.) The structure **pc_pgrpnext** is a pointer to the next **pcomm** area in the process group. The structure **pc_pgrpprev** is a pointer to the previous **pcomm** area in the process group.

Figure 10 shows a simplified view of this chain.

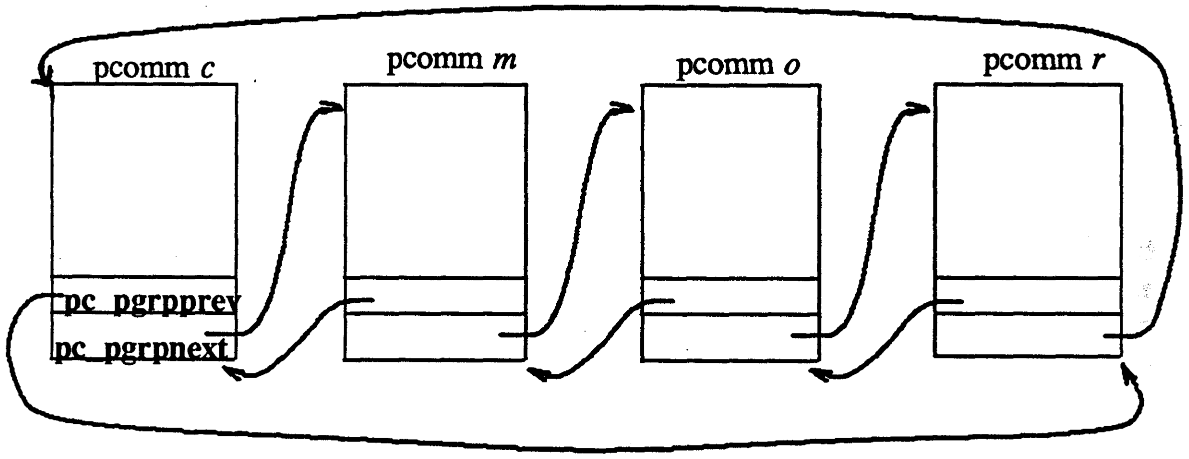

Figure 10. Chain of processes in a process group

Chain of processes in a job

This chain is a circular, singly-linked list of processes (**pcomm** areas) in a job. The job table, **jtab**[] (defined in **sys/jtab.h**), is an array of pointers, one for each job; each entry in the job table is a pointer to a chain of processes in a job.

The field **j_youngest** in the **jtab** structure is a pointer to the youngest **pcomm** area in the job. The structure **pc_jlink** is a pointer to the next oldest process (**pcomm** area) in a job. The oldest process' **pc_jlink** pointer points back to the youngest process in the job.

Figure 11 shows a simplified view of this chain:

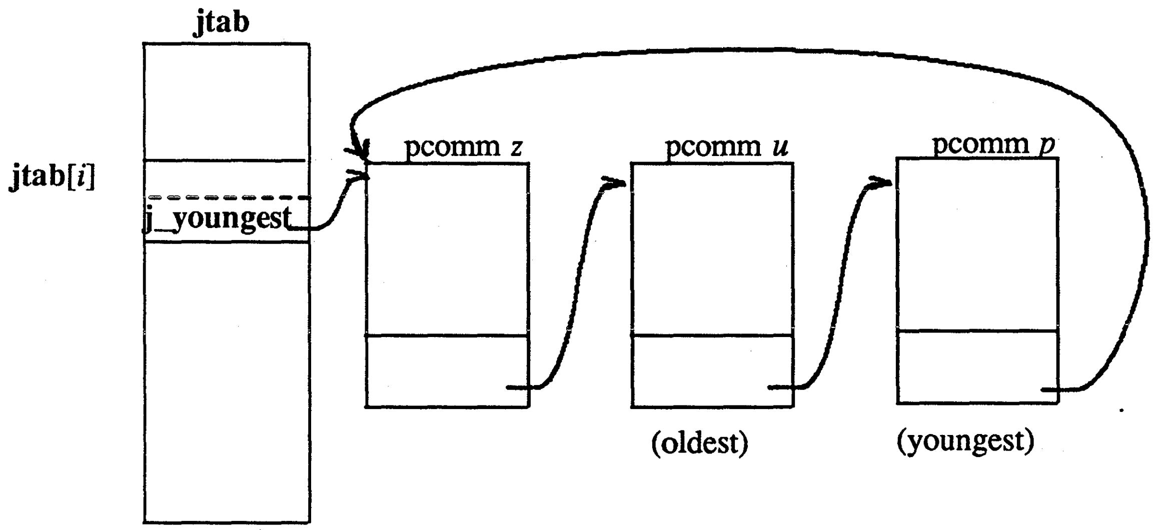

Figure 11. Chain of processes in a job

Multitask group pointer

The structure **pc_mproc** is a pointer to the process table entry of the oldest process in a multitask group.

The oldest process in a multitask group is treated specially by several routines (for example, the checkpoint/restart routines). See "Multitasking," page 97, for more information.

Figure 12 shows a simplified view of this pointer.

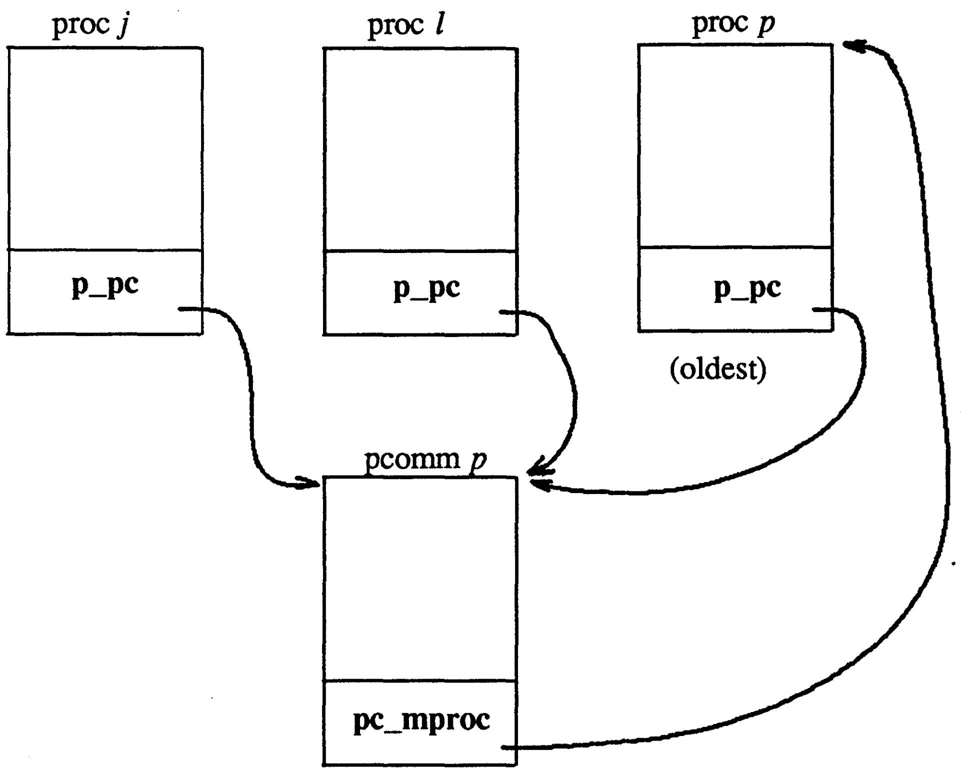

Figure 12. Multitask group pointer

Examples

The following example shows how to find a particular process table entry when the PID is known (that is, by using the value in the **p_pid** field):

```
p = pidhash[PIDHASH( search_pid )];
while( p != NULL && p->p_pid != search_pid ) {
      p = p->p_pidhash;
}
if( p == NULL )
   /* pid does not exist */
```

The following example shows how to walk through the process table (examine all active processes), excluding process 0 and zombie processes:

```
for( rp = allproc; rp != &proc[0]; rp = rp->p_next ) {
    /* no need to check for SZOMB */
    /* they are guaranteed to exist in the p_next chain */
    /* after proc[0] */
}
```

The following example shows how to walk through the process table, excluding only zombie processes:

```
for( pp = &proc[0]; pp != NULL; pp = pp->p_prev ) {
    /* no need to check for SZOMB */
    /* they are guaranteed not to exist in the p_prev chain */
    /* after proc[0] */
}
```

The following example shows how to walk through the process table, handling zombie processes one way and all other processes another way:

```
rp = allproc;
do {
   if( rp->p_stat == SZOMB ) {
      /* do one thing */
   } else {
      /* do something else */
   }
   rp = rp->p_next;
} while( rp != NULL );
```

For examples relating to specific searches (for example, by process group), refer to the **killm**() routine in the module **os/sys5.c**.

Multitasking

This section covers the following topics:

- Multitasking support in UNICOS
- Multitasked processes
- Multitasking system calls and kernel routines
- Limitations with multitasking
- Memory integrals used in accounting

Multitasking support in UNICOS

UNICOS has a mechanism that allows processes to use multiple physical CPUs for a single program.

The main conceptual change required to implement multitasking is termed the *multitasking group* (also called the *m-group*). A multitasking group is a circular linked list of processes that share one user execution image and swap image. UNIX assumes in several places that the swap image consists of a **user** area and then the user execution area (starting at the low-address end). This was changed in UNICOS to allow multiple **user** areas in the swap image. Every process table entry (the **proc** structure) now points to a process common **(pcomm)** structure, which contains the fields common to all processes in the multitasking group, such as swap-image address/size fields, memory residency information, and user execution image address fields.

Every swap image also has a shared area (the user common, or **ucomm**, area) where information for all processes in the group (for example, the table of open file pointers) is stored.

Multitasked processes

All processes within a multitasking group are equal and symmetric for most purposes. However, the oldest sibling process (the process created first) has some special cases associated with it: the **pcomm** structure resides in the oldest sibling's **proc** entry, and the oldest sibling is always the last process in the multitasking group to exit. A process can belong to only one multitasking group at a time. Thus, if any process in a multitasking group performs a **tfork**(2) operation, the new process belongs to its parent's multitasking group.

Multitasking routines

Two system calls were added to or modified in UNICOS to support multitasking. The **tfork**(2) system call is used to create another process within the multitasking group. The **resch**(2) system call reschedules a process to the (logical) end of the run queue (**runq**).

The following kernel routines are available to support multitasking:

Routine	Description
endmtsingle()	This routine ends single threading (see **mtsingle**()).
mtsingle()	This routine starts single threading. UNICOS temporarily has to force all the other processes in an multitasking group out of the physical CPUs to allow things like memory expansions.

Limitations with multitasking

Some system calls are not allowed for processes in a multitasking group; these include **fork**(2) and the **exec**(2) family of system calls.

Memory integrals

Memory integrals are algorithms for computing memory use; they are designed to take the unique considerations of multitasked programs into account. Use of the UNICOS memory integrals provides incentives for using multitasking programs, as they are able to gauge more accurately the load that a program requiring large amounts of memory places on a multi-CPU system.

There are three memory integrals used in UNICOS accounting calculations. All three integrals are maintained and placed in the accounting records. Memory integral 1 is preserved because it is a consistent algorithm, which is very important at some sites. Memory integral 2 is the default memory integral. Memory integral 3 may become the default in future releases. The constant **MEMINT** in the file **/usr/src/cmd/acct/acctdef.h** defines the default memory integral.

The UNICOS memory integrals are described in detail in the *UNICOS System Administrator's Guide for CRAY-2 Computer Systems*, publication SG-2019.

Fair-share Scheduler

The UNICOS fair-share scheduler, Share, is a per-user scheduler that runs on top of the standard per-process scheduler. It provides a fair distribution of resources for all users according to their allocation of shares. A *share* is a term covering elements of the UNICOS kernel (for example, memory usage) that affect the priority of a user's job.

The standard *(low-level)* scheduler in UNICOS schedules processes on a short term, per-process basis. Share is a per-user, short-term and long-term scheduler; it takes account of a user's past usage of machine resources. The system administrator sets the length of time during which usage is remembered; system usage is decayed by a factor referred to as *half life*. If, for example, the administrator sets the decay rate to 10 minutes, a user's usage is reduced by half in 10 minutes, to one-fourth in 20 minutes, to one-eighth in 30 minutes, and so on. (For more information, see the *UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP EA, CRAY X-MP, and CRAY-1 Computer Systems*, publication SG-2018.)

Share is inactive by default; it is initialized by **shradmin**(1M). Systems that boot and run the UNICOS 5.0 kernel without running this program will not notice any difference in the scheduler. However, different scheduling algorithms are being used. These different algorithms are explained in the remainder of this section.

The scheduler operates on two levels, the user level and the kernel level. The user-level operation is explained in the *UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP EA, CRAY X-MP, and CRAY-1 Computer Systems*, publication SG-2018. See also the **share**(info), **shradmin**(1M), **shrmon**(1M), **shrdaemon**(1M), **limits**(2), and **lnode**(4F) man pages.

Components of Share

The kernel has been modified to support the requirements of share scheduling. Share code is in the modules **os/share.c**, **os/limits.c**, and **os/clock.c**. A new system call, **limits**(2), provides an interface between the kernel and user levels of Share; it manipulates a kernel limits structure according to the value of its arguments. A daemon, **shrdaemon**(1M), updates usage information in the UNICOS user database (UDB) and recovers user information from unplanned system halts. An administrator command, **shradmin**(1M), allows an administrator to change the share scheduling priorities; **shrmon**(1M) provides an administrator display. In addition, **login**(1), **cron**(1), and all NQS utilities have been modified to access the information in the UDB and pass it on to the kernel. The include files **sys/share.h** and **sys/lnode.h** are also part of this feature.

The user limits structure (lnode)

Share relies on the UDB. When a user logs in, Share copies certain values from the UDB file (see **udb**(4F)) to initialize an entry for the user in the **lnode** table.

Share uses a per-user data structure known as an *lnode* (which stands for *limits node*) to store important information about each user's resources and shares. The **lnode** structure is defined in the include file **sys/lnode.h**.

All fields in the **lnode** structure are filled in with information from the UDB. This is done indirectly with the **limits**(2) system call.

The **lnode** structure is a subset of information kept in the kernel **lnode** table. The information in the **lnode** structure is updated in the UDB when an active user becomes inactive. The remainder of the information in the kernel **lnode** table is needed only when the user has active processes.

The kernel lnode table

The **lnode** structure is embedded in a larger **kern_lnode** structure (both structures are defined in **sys/lnode.h**). The kernel **lnode** table holds information on each active user (a user with at least one active process); each active user is represented by a **kern_lnode** structure. It contains all the temporary variables needed by Share to manage an active user. The **lnode** structure contains all the parameters maintained for active users on a long-term basis. The **lnode** structure is used in communication with the Share administrator programs that manage the scheduler interface to the kernel.

The field **p_lnode** in the process structure (**proc**) points to the **kern_lnode** structure for the user (the owner of the process).

Entries in the kernel **lnode** table are installed by **login**(1), which uses the **setlimits** library routine to access values in the UDB. **shrdaemon**(1M) removes "dead" lnodes when the last process for a user exits.

Share priority calculations

A user's usage value (for example, CPU usage, memory usage, and I/O) is that user's accumulation of costs, as defined by **shradmin**(1M). The usage is calculated by accumulating the charges incurred during a scheduling time period and dividing them by the square of the user's allocated shares. Thus, a user with a larger number of shares will have a usage with a faster decay rate than a user with a smaller number of shares.

The share scheduler affects the low-level scheduling of processes by calculating a normalized usage from each lnode's **usage** field; this number is added to the priority of a process. The (numerically) higher its priority, the less often a process is scheduled; therefore, processes belonging to users with high usage will get a smaller share of resources. Note, however, that at any one time, a user can use all of the resources available provided there is no competition from others.

There are three cycles for adjusting a process' priority, as follows:

- Minor cycle; this is every clock tick (1/60 of a second or 1/100 of a second, depending on the value **OS_HZ** in the file **sys/param.h**). Every minor cycle, the usage value (the **kl_usage** field in the **lnode**) is multiplied by the rate (**kl_rate**), and the result is added to its scheduling priority (**p_sharepri**); this calculation is expressed as follows:

  p_sharepri += p_lnode–>kl_usage * p_lnode–>kl_rate

  The value in **p_sharepri** is decayed (in the routines in **os/clock.c** and **os/share.c**) by an amount that depends on the process' nice value (the value in the **p_nice** field); the lower the priority of the process, the slower the decay. This value is copied into the low-level scheduler's priority (**p_pri**) whenever the process is run in user space.

- Middle cycle; this is every second. Each middle cycle, the priorities of all the processes on the run queue are re-evaluated; their resource consumption values are decayed at a rate determined by their nice values.[†] This has the effect of gradually moving processes to the top of the queue and assures that every process will be scheduled to run. It is necessary to balance the rate of migration to the top of the queue with the rate of resource consumption so that relative priorities are remembered for a long enough time to prevent large numbers of processes from migrating to the top of the queue.

- Major cycle; this is every 4 seconds. Every major cycle, overall resource usage is decayed for all processes in the **lnode** table. A major cycle is 4 seconds in the released system; the administrator can change the length of the major cycle with the **shradmin**(1M) command.

[†] Processes with nice values of 0 move toward the top of the queue faster than those with nonzero values.

Kernel-level support functions

The following modules contain the Share routines:

Module	Description
os/share.c	This module contains the major cycle code (the code executed once every major cycle).
os/limits.c	This module contains the system call used by the user-level Share code to interrogate and modify the kernel **lnode** table. The routines in **limits.c** look at the **lnode** table to determine the limits for a user.
os/clock.c	The routines in this module have been modified to handle the new calculations for each minor cycle (clock tick), each middle cycle (1-second interval), and each major cycle.

When the current process voluntarily gives up the CPU, or a scheduling cycle is initiated by the kernel (the minor cycle), process priority of the current process is evaluated, and its priority is adjusted based on its relative position among the available processes and the proportion of shares allocated to the user. This may lower, raise, or leave unchanged the position of the process in the scheduling queue. When this evaluation has been done, the *n* processes (where *n* is the number of active CPUs) at the top of the queue are started.

At the kernel level, processes are scheduled on a priority-ordered queue; the process with the highest priority (that is, the priority with lowest numerical value) is at the front, or top, of the queue. The user's entry in the **lnode** table keeps track of the count (or rate) in the **kl_rate** field; this is the number of processes that the user has on the queue. The value in **kl_rate** is decayed over time.

At every clock tick, the current process has its priority incremented by an amount proportional to the values in the **kl_usage** and **kl_rate** fields in its owner's kernel **lnode** table entry.

Every second, the priority of each process is decayed by an amount depending on the nice value for the process (the nicer the process, the slower the decay).

Each process is also charged at various times for consumption of resources, by an amount appropriate to the resource, which is added into the **kl_cost** field in the kernel **lnode** table entry.

Idle processes

Idle processes are considered users (with the name **Idle**) of system resources. Therefore, the idle processes must have an entry in the UDB; user number eleven (11) has been reserved for representation of the user **Idle**'s usage of the system. Share guarantees that these processes never interfere with other processes by attaching the idle processes to a special kernel **lnode** table entry belonging to the user **Idle**; this is done at boot time. (The program **shrdaemon**(1M) executes a special system call (the routine **limits**() in **os/limits.c**) during start-up to change the boot-time identity (UID of -1) of the user **Idle** to one represented in the UDB.)

Note that the idle processes must be allocated zero shares so that an idle process runs only when no other user requests the CPU.

The usage figure for the idle processes is set arbitrarily large by the user-level portion of Share (this is actually handled in **os/main.c** and **os/share.c**); this ensures that idle processes stay at the bottom (the low-priority end) of the active queue.

Effect of nice(2)

The **nice**(2) system call has a slightly different effect under Share. The nice parameter for a process now affects the rate at which its priority decays to a higher priority over time. "Nicing" a process will make it run more slowly, by reducing its effective share of the resources, but it may not run more slowly than another user's processes if that user has an even lower effective share of the resources. However, processes with a nice priority of 19 are guaranteed to run only when no other processes need the CPU. "Niced" processes are charged less for CPU time than normal processes; processes with a nice value of 19 are charged almost nothing for CPU time.

Recovery

UNICOS recovery provides the ability to stop and save a process, a multitasking group, or a job and restart it later. This section describes how recovery (also called *job and process recovery*) has been implemented in the UNICOS kernel.

There are several reasons for providing a recovery mechanism in UNICOS:

- Many applications require more time than can be delivered in one continuous interval. Machine maintenance, system reconfiguration, and the need to distribute machine resources for optimum use by a demanding user community can prevent a site from dedicating the machine for a time that is long enough for the most compute-bound applications to complete their work.
- Applications can insure work by saving partially completed work at critical points or plateaus; this guards against unexpected events (machine and program failures).

The subsections in this section cover the following topics:

- Definitions of terms used in describing recovery
- Issues in checkpointing and recovery
- Description of restart files and their structure
- Description of the checkpoint algorithm
- Description of the recovery algorithm

Definitions of terms

This subsection provides term definitions and references used in describing recovery.

Term	Description
chkpnt(2)	System call responsible for *freezing* (temporarily halting the execution of) the target process set and then placing all of the information necessary to restore the target processes into a restart file.
job	Collection of one or more processes created with **fork**(2) that are tracked as a single unit, through a job ID (JID), for the purposes of resource control, accounting, and recovery.
Multitasking group	Collection of processes created with **tfork**(2); the processes in a multitasking group share many resources, including a common address space. A multitasking group is sometimes referred to as an *m-group* or *tightly coupled processes.*
restart(2)	Inverse of the **chkpnt**(2) system call. The **restart**(2) system call accepts a restart file as input and then recovers the processes described by the restart file.
restart file	File created by **chkpnt**(2) that contains all information needed to restore the target process set to its execution state when the restart file was created. A restart file is sometimes called a *recovery image.*
restarted process	Process that has been recovered from a restart file.
target process set	Single process, multitasking group, or job.

Issues in checkpointing and recovery

This subsection discusses some issues in checkpointing and recovery.

The system prohibits attempts to checkpoint processes, multitasking groups, or jobs that have no hope of recovery. For example, a process with an open socket connection to another process on a remote machine is not checkpointable, because there is no means to checkpoint and restart the target process and its remote network peer in a consistent state. Similarly, a process cannot be checkpointed if it has an open pipe connection to another process that is not also included in the set of processes being checkpointed.

UNICOS recovery differentiates between *fast* and *slow* I/O operations. *Fast I/O* is I/O to fast devices, such as memory and disks; *slow I/O* is I/O to slow devices, such as terminals and pipes. When a process is checkpointed, all fast I/O is allowed to complete prior to the actual checkpoint operation (before the restart file is written), because all fast I/O completes relatively soon. Slow I/O operations in progress when the checkpoint operation starts are immediately frozen; they are restarted when the process is recovered. All outstanding slow I/O requests are recorded in the restart file for later recovery.

In general, slow I/O operations in progress at the time of checkpoint are always restarted upon recovery as though no checkpoint/restart activity occurred. However, slow I/O operations started with the **listio**(2) system call are not restarted upon recovery; all such I/O requests fail with the **EINTR** error upon recovery.

Only the super user is allowed to checkpoint and restart entire jobs. Because many resource consumption limits are enforced at the job level, this restriction prohibits ordinary users from using the Network Queueing System (NQS) to escape their resource limits (by submitting NQS batch requests, checkpointing the job created to service the request, and then restarting the job as part of an interactive session).

The system makes extensive efforts to prevent the recovery of a target process set if the recovery could lead to the production of incorrect results. Several validity checks are performed on each file referenced by the corresponding restart file whenever a restart operation is attempted; if any of the referenced files have been changed, the restart operation fails. However, **restart**(2) has a flag, **RESTART_FORCE**, to restart the necessary processes even if one or more of the needed files has been modified.

Checkpoint and restart activity is transparent to the target process set. However, applications are informed that they have been restarted from a restart file. Upon the successful completion of a restart operation, the **SIGRECOVERY** signal is sent to one member of each recovered multitasking group. The **SIGRECOVERY** signal is ignored by default, but a process can catch this signal if desired.

Process IDs (PIDs), process group IDs (PGIDs), and job IDs (JIDs) are preserved across checkpoint and restart operations. However, maintaining the ID values creates a problem if a needed ID value is already in use by another process or job when a restart operation is attempted. This problem remains unresolved; if the situation occurs, the restart operation fails.

Restart files

A restart file is constructed by the **chkpnt**(2) system call whenever a target process set is successfully checkpointed. The file contains the information needed to restore the checkpointed processes to their original state.

Restart files are created with all write permissions disabled; the owner is allowed read permission only if no setuid processes were checkpointed into the file. (A restart file containing any setuid processes is not readable; this prevents sensitive information from being read by unauthorized users.)

A restart file is a regular file (type **IFREG**). However, it is distinguished from other types of regular files by a bit in the **i_ftype** field of the i-node; this bit is called the *restart file* *attribute bit* (**IRESTART**, defined in the include file **sys/inode.h**). This bit allows the system to prevent attempts made at changing the protections or ownership of any restart file. There is no way to set or clear the restart file attribute bit in a file, and there are only two ways to create a restart file with this bit set:

- Using **chkpnt**(2) to checkpoint a target process set.
- Using the **open**(2) system call with **O_CREAT** and **O_RESTART** to create a new file with the restart file attribute bit set. The **restore**(1M) utility uses **open**(2) in this manner to restore restart files. Only the super user can use this method of creating a restart file.

In the 4.0 release of UNICOS, the super user could use the **mknod**(2) system call to create a file with the restart attribute bit set. This is no longer possible in the 5.0 release.

Unlinked temporary files of limited size are supported by the UNICOS 5.0 version of recovery. Unlinked temporary files always disappear as soon as a process exits. To circumvent this difficulty, the contents of the unlinked temporary files are copied into the restart file during checkpointing; the unlinked temporary files are then rebuilt from the saved copy during recovery. The parameter **MAX_UNLINKED_BYTES** (defined in **sys/param.h**) defines the maximum amount (the number of bytes) of unlinked file data that can be placed in the restart file; this prevents restart files from becoming too large.

Restart file structure

The restart file structure is defined in the include file **sys/restartfil.h**. The include file **sys/restart.h** contains the definitions for the restart specification flags, which are used in the **chkpnt**(2) and **restart**(2) system calls.

As shown in figure 13, a restart file is divided into several sections.

Header
I-node descriptors
File table descriptors
File lock descriptors
I/O header descriptors
I/O request descriptors
Pipe and unlinked file data
Target process images

Figure 13. Restart file structure

The sections contain the following information:

Header
The first section of a restart file contains information describing the contents of the restart file. Fields in the header define the type of entity contained in the restart file (for example, a process or job) and the number of various object descriptions appearing in the file (for example, the number of i-node, file table, file lock, I/O header, and I/O request descriptors). Other information present in the header includes the number of processes and multitasking groups defined in the restart file and the characteristics of any original controlling tty connection.

The format of the header section is described by the **reshdr** structure in **sys/restartfil.h**.

I-node descriptors
This section of a restart file contains descriptions of each file in use by the checkpointed processes. Each entry in this section contains the information needed to recover the file that it describes (for example, the file's mount device, file type, owner ID, user ID, group ID, i-node number (i-number), and i-node generation number). If the i-node describes a pipe, its entry in

this section also contains a pointer to the pipe data in the pipe and unlinked file data section of the restart file (as a byte offset relative to the start of this section).

The format of the i-node descriptor section is described by the **resinodesc** structure in **sys/restartfil.h**.

File table descriptors

This section of a restart file contains a descriptor for each open file in use by the checkpointed processes. A file table descriptor completely describes a file table entry in use by one or more of the checkpointed processes. This information includes the file table flags (for example, read, write, or append), the index of the i-node descriptor for the file (this is a pointer to the corresponding i-node data, stored as an index relative to the beginning of the i-node data section of the restart file), and the current file position (the byte offset from the beginning of the file).

The format of the file table descriptor section is described by the **resfildesc** structure in **sys/restartfil.h**.

File lock descriptors

This section contains a descriptor for file locks in use by the checkpointed processes. The descriptor contains the i-node description index (the index of the i-node to which the lock was applied) and the file lock, which is a description of the original lock.

The format of the file lock descriptor section is described by the **resflockdesc** structure in **sys/restartfil.h**.

I/O header descriptors

This section of the restart file contains the common I/O header structures (**ioh** structures) in use by the checkpointed processes. A user-level I/O request may be broken into smaller requests that each reference a common I/O header. Each such coordinating structure combines smaller I/O requests and has a corresponding descriptor in this section of a restart file.

The format of the I/O header descriptor section is described by the **resiohdesc** structure in the include file **sys/restartfil.h**.

I/O request descriptors

This section of the restart file contains a descriptor for each user-level I/O request in use by the checkpointed processes. Each I/O request descriptor contains the information necessary to restore the associated I/O request. This information includes the index of the i-node descriptor describing the file for which the I/O request is taking place, the index of the I/O header coordinating the request, and a description of the saved I/O request. Multiple I/O request descriptors can reference the same I/O header.

The format of the I/O request descriptor section is described by the **resiodesc** structure in **sys/restartfil.h**.

Pipe and unlinked file data

This section of the restart file contains the data present in pipes and unlinked temporary files that are referenced by the checkpointed process.

The data in this section is not in a predetermined format. The size of this section is recorded in the **rh_nobjdata** field in the header section of the restart file.

Target process images

This section of the restart file contains the images of the target process set. Every multitasking group that has been checkpointed into the restart file appears here. (Note that a single process is considered a degenerate form of a multitasking group; see "Multitasking," page 97, for more information.)

The format of a single multitasking group description in this section is as follows:

Process (**proc**) structures for all processes in the multitasking group, oldest process first
User (**user**) structures for all processes in the multitasking group, oldest process first
User common (**ucomm**) structure for the multitasking group
Text and data for all processes in the multitasking group, oldest process first

If necessary, the system rounds up to the next sector boundary between each set of entries in a multitasking group section.

The number of multitasking group images is recorded in the **rh_nmtasks** field in the header section of the restart file.

The checkpoint algorithm

The **chkpnt**(2) system call is used to checkpoint a target process set. It locates and freezes the target process set, then writes the restart file, so that the target process set can be recovered later with the **restart**(2) system call.

There are three phases of a **chkpnt** system call: locating the target process set, freezing the target process set, and writing the restart file. These three phases are described in the following subsections.

Phase 1: Locating the target process set

The first phase of checkpointing consists of the following operations:

1. Locating the target process set
2. Performing ownership checks on the target process set
3. Checking for zombie processes

These operations are described in the following subsections.

Locating the target process set

The target process set is identified by *category* and *id* arguments to the **chkpnt**(2) system call. *category* is either **C_PROC** or **C_JOB**, indicating that a process or a job, respectively, should be checkpointed. The *id* argument identifies the PID of the process or JID of the job, as appropriate.

When **chkpnt** is invoked to checkpoint a specific process, it also checkpoints all of the sibling processes in the multitasking group associated with that process. This is done because any restart operation is meaningful only if all processes within a multitasking group are also recovered upon restart.

Performing ownership checks

After the target process set is located, **chkpnt** checks the process ownership to ensure that the caller owns the processes that are to be checkpointed. (Of course, the super user can checkpoint processes or jobs owned by other users.)

Checking for zombie processes

A final check is made to ensure that the target process set contains at least one active (*nonzombie*) process. If all of the targets are zombie (exiting) processes, the checkpoint operation fails, because it is useless to checkpoint an entire set of processes that have finished execution. However, when a job is checkpointed, any zombie processes present in the job must be included in the checkpoint, because the parent process will likely issue a **wait**(2) system call to retrieve the status of the zombie child process.

Phase 2: Freezing the target processes

The second phase of **chkpnt** is concerned with freezing the target process set. The system cannot copy the processes' text and data into the restart file until this is done, because the processes might change state while the copy is being made.

A process may be frozen quickly or slowly. The determining factors are as follows:

- If the process is not executing a system call, it is frozen quickly. If the process is connected to a CPU executing user code, the CPU is immediately yanked away. The **P_FROZE** flag in the process structure and the **PC_FREEZE** flag in the process common (**pcomm**) area are set so that the process is blocked from further execution.
- If the process is executing a system call, but is sleeping at an interruptible priority, it is also frozen quickly. The process is blocked from execution as in the first case.
- If the target process is executing a system call and is sleeping at a non-interruptible priority, a further check must be made. The **P_RWAIT** flag in the process structure is checked; if it is set, the process is frozen quickly as in the first two cases. If the flag is not set, the process is frozen slowly; the **PC_FREEZE** flag in the process structure tells the process that when its

system call work is done, it should block itself from further execution and wake up the checkpointing process when it has done so. The checkpointing process then waits for this to occur.

In practice, all of the processes that cannot be immediately frozen are told to block themselves when their critical work is complete, before **chkpnt** does any waiting. A second loop in the algorithm then waits for all of the processes not yet blocked to block themselves.

When all of the processes are finally blocked, the checkpointing process waits for all fast I/O operations (for example, disk operations) associated with the target processes to complete (the **pc_fastiocnt** field in the **pcomm** area is used to keep track of outstanding fast I/O operations). All slow I/O operations are recorded in the restart file.

When all fast I/O is complete, all of the processes in the target process set are completely frozen.

Phase 3: Writing the restart file

In the third phase of checkpointing, all state information that is necessary in order to recover the target process set is written to the restart file. The system examines each process, gathering information about file i-node, file table, and pipe usage. During this examination, checks are made to see that none of the processes is using any irrecoverable resources. (If one or more is, the checkpoint operation is terminated.)

A check is also made at this point for any irrecoverable situations. The system cannot checkpoint processes with pipes going outside the target process set, processes with open socket connections, or processes with more than the system-defined amount of open unlinked temporary file data (default of 1 Mbyte in the released system).

The restart file is written as follows:

1. The system writes the restart file header in the format described in "Restart file structure," page 113. The header contains all of the resource usage counts, the description of any associated terminal connection characteristics, and job resource consumption statistics (if an entire job is being checkpointed).
2. The system writes each section of the restart file in sequential order. The system maintains the appropriate i-node, file table, and I/O header descriptor indices, in order to correctly record the dependency relationships of the descriptors in the later descriptor sections. For example, when the file table descriptors section is written, the system uses the mapping (of the i-node and file table entries to the index of the corresponding i-node descriptor in the restart file) that was built when the i-node descriptors section was written.
3. The system writes the target process images as the last section of the restart file. For each multitasking group, the system loops over all of the processes, from oldest to youngest, writing out their process and user structures in the order described in "Restart file structure," page 113.

   A single **ucomm** structure is then written for the multitasking group. The **ucomm** structure is changed slightly when it is written in the restart file; all file and i-node pointers are recast as the integer indices of the corresponding file and i-node descriptors.

   After the **ucomm** structure is written, the system writes the multitasking group's execution image in the restart file.

When all of the multitasking group information has been written to the restart file, the restart file is complete. If the **CHKPNT_KILL** flag was specified in the call to **chkpnt**, all of the checkpointed processes are killed. Otherwise, all flags blocking the execution of the checkpointed processes are cleared, and the processes resume execution.

The recovery algorithm

The **restart**(2) system call restores the processes described in a restart file to their saved execution state. The recovery process has two distinct phases: initial restoration and completed restoration. These phases are described in the following subsections.

Phase 1: The initial restoration

The first phase of recovery consists of the following operations:

1. Reading the header section of the restart file
2. Creating a restart prototype process (RPP) for the oldest process of each multitasking group
3. Creating an RPP for the remaining processes in each multitasking group
4. Partially recovering each process

These operations are described in the following subsections.

Reading in the header

The recovery operation begins by opening the restart file and reading in the header section. Sanity checks are conducted on the header to detect corrupt restart files. If all checks are passed, the information in the header is used to compute the byte offset of the descriptor sections in the rest of the restart file.

Creating the RPP

The system reads in the process table entry (**proc** structure) of the oldest process in the restart file. The system executes the kernel fork routine, **dofork**(), to create an RPP for this first process.

This fork operation is unusual for two reasons: the first RPP is of zero size, because its text and data size are not known at this point (they are later read from the restart file); and the first RPP has the same PID and PGID the original process had when it was checkpointed. (If any of the ID values is already in use, the recovery operation fails.)

The process that started **restart** then goes to sleep until the first phase is completed. The first RPP checks to see if an entire job is being recovered. If this is the case, the first RPP places itself into a new job, using the original JID. (As with the other ID values, if this JID is in use, the restart fails.)

Creating RPPs for other processes

The first RPP process reads through the remainder of the restart file. It uses **dofork**() to create an RPP for the oldest process of each multitasking group (as before, ID values are preserved, and the processes are of zero size).

Each of these RPPs calls the kernel routine **resmtask**() (in **os/restart.c**) to restore the other processes in its multitasking group and to restore the user common (**ucomm**) area for the multitasking group. The first RPP also calls this common routine to restore its multitasking group contents.

Partially recovering each process

When RPPs for each process in the restart file have been created, each RPP recovers its own **user** structure from the restart file. Then the oldest RPP for each multitasking group restores the information common to the entire multitasking group. This part of the recovery is quite intricate, because it involves the restoration of all of the i-node and file table entries in use by the multitasking group, plus the possible reconnection to a controlling tty if the recovered multitasking group had an original tty connection. Recovering the i-node and file table entries requires some care, because the needed i-node or file table entry may have already been recovered by another related multitasking group that has already completed some portion of its recovery. To prevent duplicate recoveries, tables are maintained internally to keep track of i-node and file table references already restored.

The recovery of i-nodes must also be done carefully. Many integrity checks are done to ensure that the recovered i-node truly describes the file referenced by the processes being recovered. If the file has been changed, the restart operation must fail, unless the **RESTART_FORCE** flag was specified in the options argument to the **restart** system call.

The recovery of pipes is also complicated, because data present in the pipe at checkpoint time needs to be restored. The first RPP to recover a file table entry that refers to a pipe i-node is responsible for writing the data into the pipe. Modifications made to the pipe driver allow this activity to occur even when no other process has the pipe open.

After the RPP recovers i-node and pipe data, it restores its own **user** structure, expands the multitasking group to its original size, and then recovers the multitasking group text and data from the restart file.

Phase 2: Completing the restoration

The RPPs cannot completely restore themselves, so the process that originally called **restart** handles the second phase of the recovery. It performs the following operations:

1. Reads all the file lock descriptors from the restart file. For each file lock, recovery information is placed at a fixed address, and the RPP that originally owned the lock is awakened to restore its own file lock with the internal file locking system call.
2. Reads all the I/O request descriptors and I/O header descriptors from the restart file and reissues all I/O requests that were outstanding when the processes were checkpointed.

3. Notifies all recovered processes when the I/O recovery is complete; these processes go to sleep again after advancing to the next state-change wait loop.
4. Reads through the multitasking group descriptions of the restart file to restore all process and job table information. Any terminal connection characteristics are also restored at this time.
5. Clears the kernel flag (the **PC_RSTART** flag in the **pcomm** area) for each recovered process; these are the flags that have kept the RPPs from being seen by the rest of the system. (The RPPs are now real processes.)

The recovered processes are then awakened; each process does some final restorative work (resetting the system call arguments and timing statistics) before reentering the system call thread of the kernel. Each process either restarts the system call that it was in when originally checkpointed or returns to the execution of the recovered user program.

I/O Management

UNICOS I/O

This section describes UNICOS I/O on CRAY-2 systems. It covers buffered and raw I/O, synchronous and asynchronous I/O, and the **reada**(2) and **writea**(2) system calls.

Buffered I/O, also called *block I/O*, is the default form of UNICOS I/O. With a buffered I/O transaction, data is moved through system buffers between the device and the user's buffer. Raw I/O is available by specifying the O_RAW bit to the open(2) system call. With a raw I/O transaction, data is moved directly between the user's buffer and the specified I/O device. When raw I/O is used, the user process is locked in memory for the duration of the operation (that is, the process cannot be swapped).

With synchronous I/O, the calling process sleeps until the requested data is available from the device. Asynchronous I/O returns control to the calling process after issuing the I/O request. Notice of completion is through an I/O status word and an optional signal.

The **reada**(2) and **writea**(2) system calls were added to UNICOS to support asynchronous I/O. The **reada**(2) system call performs an asynchronous read operation from a file; the **writea**(2) system call performs an asynchronous write operation from a file.

The file position for the read or write operation is always the current position at the time of the **reada** or **writea** call. At that time, the file's position is incremented by *nbytes*. In this way, **reada**, **writea**, and **lseek**(2) operations can be interspersed, and the file position is incremented naturally.

In most cases, it is undesirable for any I/O completions to go unnoticed. To ensure that this does not happen, the following must be taken into account. All outstanding I/O operations must have their own status words so that the user program can tell which have completed. One or more signal numbers can be used for I/O completions, but each signal should have its own handling routine. When an I/O completion handler is entered, the status words under its control should be scanned for completed I/O operations. As the status words are processed, they should be zeroed. At the end of I/O completion handling, the status words should be rescanned for newly completed I/O operations. If more are found, control should loop back so they are also processed. Otherwise, the handler will be exited with a **SCTL_RET** action (with the **sigctl**(2) system call). UNICOS then checks the last completion status word to make sure it was serviced, and, if it is still nonzero (that is, not serviced), the signal handling routine is reentered.

Disk Striping

This section discusses foreground striping, which is also called *disk striping*, and background striping.

Foreground striping

Foreground striping refers to the practice of combining two or more physical disk devices for the purpose of increasing the disk transfer rate.

A striped disk device, also known as a *stripe group*, is comprised of from 2 to 32 physical disk devices, also known as *members*, logically tied together by the foreground processor (FP) and the disk driver.

Track I/O to a striped device is done in units of striped tracks. The size of a striped track is the size of one physical track times the number of members of the stripe group. I/O of less than a stripe track in size is done in the background disk driver, using requests to the individual physical members of the stripe group. Note that all drives in a stripe group must be of the same device type.

A stripe group is defined by the hardware configuration file **cf/hconf.h**. It has an entry in **devblock** in low common memory (**md/lowcm.s**). (See "Foreground Processor and Kernel Communication," page 33, for more information on **devblock.**) A list of FP index numbers of the members of a stripe group make up one entry of the **stripeblk** table defined in **md/lowcm.s**. The **stripeblk** is defined as a character array and is used by the stripe disk driver, **io/sdsk.c**, for mapping non-stripe sized requests and raw disk requests to the individual physical members of the stripe group. Currently, the maximum allowable number of stripe groups is 4. There can be no common members among the stripe groups.

Foreground striping is most useful for swapping, but a foreground striped file system can be built as long as a track buffer pool of the appropriate stripe track size is selected in **sys/param.h**. See the *UNICOS System Administrator's Guide for CRAY-2 Computer Systems*, publication SG-2019, for more information on configuring foreground striped file systems.

A striped swap device or striped file system can use all or part of the defined striped group. The device i-nodes determine the starting and ending blocks for a given partition. The partitions are identical for each member of the stripe group. Individual members of a striped file group can contain non-striped file systems. If this is done, the individual file systems should be backup file systems or file systems that are not heavily used at the same time as the stripe group.

The blocks in the first cylinder of a two-member DD-49 stripe partition are numbered as follows:

Member		0	1
Head	0	0-41	42-83
	1	84-125	125-167
	.		
	.		
	.		
	7	546-588	630-671

A device i-node of a two-member DD-49 stripe partition that is one striped cylinder in length, starting at cylinder 1, appears as follows:

```
M0:     character special (0/200)    4 32 672 672 42 8 2 0
```

The striped disk driver needs minor device numbers for communication with the individual members of the stripe group. These minor numbers are assigned consecutively. The minor numbers in the above example would be 201 for member 0 and 202 for member 1.

Background striping

Background striping refers to making consecutive allocations for a single file on consecutive partitions of a cluster.

A disk cluster allows several disk partitions to be mounted together on a single mount point as one file system. The partitions need not be the same size. Files can overflow from one partition to another in the same cluster.

In file systems on many UNIX systems, disk partition sizes are fixed in the driver with constants and are selected with the three low-order bits of the minor device number. This limits the system to a small number of fixed partition sizes. If a disk has a bad sector, partitions must be made to skip it. With this restriction, it is quite easy to run out of partitions and difficult to change them. Also, each partition is an independent file system, so files are limited in size.

In UNICOS, major and minor device numbers do not specify a partition's characteristics (controller, drive, and partition) but simply a logical partition number. The information about the partition (type, channel, controller, unit, start, and length) is stored in the i-node of the special file representing the device. Bad blocks are handled independently of partitioning.

When a file is created, the **cbits** (*cluster bits*) field allows the user to specify which partition or partitions of the cluster should be used. Each bit of this field (starting with 2^0) corresponds to a disk partition in the cluster; the order of the partitions is the order in which they were mounted. If the user specifies 0011 for one file and 1100 for another, each is striped two drives wide on different drives. Bits beyond the number of drives in the cluster are ignored, so full-width striping can be easily specified with **–1** (all bits set). If **cbits** is 0, the system chooses a partition.

In UNICOS, pointers within the file system were expanded to allow clusters. I-node pointers have a device field and a i-node number field; block pointers have a device field and a block number field. Each entry in the **c2mount** table corresponds to a disk partition. The partitions are linked together for easy conversion of **cbits** to partitions during track allocation.

Miscellaneous Information

Security

This section describes the kernel-level include files and routines for the UNICOS security feature. It covers the following topics:

- Changes to the UNICOS kernel
- Additional and changed include files
- Security system calls
- Additional and changed kernel routines

For information on the user-level security features, see the *UNICOS Security Administration Reference Manual*, publication SR-2062. For information on security definitions and requirements, see the following publications:

> *UNIX System Security*
> Patrick H. Wood and Stephen G. Kochan
> Hayden Book Company, 1985.
>
> *Department of Defense Trusted*
> *Computer System Evaluation Criteria*
> CSC-STD-001-83
> Library No. S225,711
> 15 August 1983

Changes to the UNICOS kernel

The *security kernel* executes discretionary access controls and mandatory access controls required for implementation of the UNICOS security policy. This security kernel resides completely within the UNICOS kernel. All discretionary access and mandatory access controls for file objects are in the routine

c2access() in the module **uts/fs/c2/c2subr.c**; all mandatory access controls for TCP/IP objects (sockets) reside in the routine **net_access**() in the module **tcp/kernel/sys/uipc_sysca.c**.

The security system calls establish and change the user's security policy, alter object markings (labels and compartments), and maintain the access control list (ACL) assignments. These system calls are implemented in the kernel; their kernel entry points are in the module **uts/os/secure.c**. Other new security functions, such as ACL searches and console access constraints, are implemented as single routines and are also in **secure.c**.

Other security policy software, such as that for permission checks, directory security hierarchical control, file creation labels, file system mount and unmount operations, file mode and owner changes, and file removal, is implemented in the UNICOS kernel function that controls the relevant operation.

The security software is activated by setting the **SECURE_SYS** and **SECURE_INC** parameters in the include file **sys/param.h** to 1. This directs the compiler to compile the security code bounded by the directives **#if SECURE_SYS** and **#endif SECURE_SYS**.

The design philosophy for the UNICOS security feature is to maintain UNICOS usability and to preserve the UNICOS operating system, which runs on all Cray systems. The security extension to UNICOS retains the security mechanisms and user interfaces present in the UNIX operating system. Modifications and extensions to UNICOS have been confined, for the most part, to small kernel functions; access controls for all processes are centralized and exercised consistently with the existing operating system software.

Include files

The following include files were added to support the UNICOS security feature:

File	Description
sys/acl.h	Defines the contents (format) for an access control list
sys/nal.h	Defines the contents for a network authorization list (NAL)
sys/secparm.h	Defines the security parameters for a secure system
sys/secstat.h	Defines status format for security label information (returned by **secstat**(2))
sys/slog.h	Defines the structure of the security log pseudo device, **/dev/slog** (see **slog**(4D) for more information)
sys/slrec.h	Defines the security log entry formats (see **slrec**(4F) for more information)
sys/usrv.h	Defines the format of the **usrv** structure, which contains the user's security parameters.

The following include files contain modifications to support the security feature:

File	Description
sys/filsys.h	Includes the file system's lower security level, upper security level, and the **SECURE** magic number.
sys/ino.h	Includes security level and compartment label information and access control list disk address for the in-core i-node.
sys/inode.h	Includes the security level and compartment label information and access control list disk address for the disk i-node.
sys/proc.h	Includes active security level and active compartments for a process.

File	Description
sys/unistd.h	Defines the system configuration request **SC_CRAY_SECURE_SYS**. When this request is used as the argument for the **sysconf**(2) system call, the value 1 is returned if the system was built as a secure system; otherwise, the value 0 is returned.
sys/user.h	Includes a user's lower security level, upper security level, authorized compartments, and permissions. Also defines the user's maximum security level, active security level, active compartments, permissions, and the security level of open file(s); these are needed in the **ucomm** area to validate file access and other system services.

Security system calls

This subsection describes the system calls added to UNICOS in order to implement the security feature. See the *UNICOS Security Administration Reference Manual*, publication SR-2062, and *Volume 4: UNICOS System Calls Reference Manual*, publication SR-2012, for information on the use of these system calls.

These system calls are defined as follows:

System call	Description
fsecstat(2)	Gets file's security levels and compartments (usage: general user)
getfacl(2)	Gets access control list assigned to a file (usage: file owner or security administrator)
getfcmp(2)	Gets compartments of a named file
getflvl(2)	Gets security level of a named file
getsysl(2)	Gets system security levels (usage: general user)

System call	Description
getucmp(2)	Gets user's active compartments
getulvl(2)	Gets user's security level and compartments (usage: general user)
getusrv(2)	Gets user's minimum/maximum security levels, valid compartments, and permissions (usage: users and subsystems)
rmfacl(2)	Removes access control list assigned to a file (usage: file owner or security administrator)
secstat(2)	Gets file's security levels and compartments (usage: general user)
setfacl(2)	Assigns access control list to a file (usage: file owner or security administrator)
setfcmp(2)	Sets file's compartments (usage: security administrator)
setflvl(2)	Sets file's security level (usage: security administrator)
setsysl(2)	Sets system security levels (usage: security administrator)
settfm(2)	Sets the trusted facility management (TFM) mask (usage: user with TFM privileges)
setucmp(2)	Sets user's compartments (usage: general user)
setulvl(2)	Sets user's security level (usage: general user)
setusrv(2)	Sets user security level, compartments, and permissions (usage: trusted subject process)
slgentry(2)	Makes security log entry (usage: trusted processes)

Kernel routines

The following kernel modules and routines have been added to support the security feature:

Module	Description
os/secure.c	This module contains all system calls and supportive functions for the security feature. Some important routines are as follows: **aclchk**() Performs discretionary access checks against an access control list. **secure_console**() Verifies that the system administrator or security administrator has logged in at the designated administration console (defined in **sys/secparm.h** as **SECURE_SYSTEM_CONSOLE** and **SYSTEM_ADMIN_CONSOLE**) and that the system operator has logged in at the designated operator's console (defined in **sys/secparm.h** as **SECURE_OPERATOR_CONSOLE**). **secure_filsys**() Verifies that a file system to be mounted has lower and upper security levels that fit within the security window of the system. **secure_label**() Compares the user's active security level against the file's security level and checks that the file's compartments are equal to or a subset of the user's active compartments.
os/slogext.c	This module contains security log functions available to users and the system. These routines include **slginit**(), **slgentry**(), **slggo**(), **slgeoj**(), **slgtchg**(), **slgcdr**(), **slgput**(), **slgcdr**(), **slgput**(), and **slglogn**().

The following kernel modules have been modified to support the security feature:

Module	Description
fs/c2/c2alloc.c	The routine **c2alloc**() ensures that the file system has the **SECURE** label and that the user's active security level and compartments are acceptable for those authorized for the file system. **c2alloc**() also assigns the user's active security level and compartments to the file's i-node.
fs/c2/c2iget.c	The routine **c2iput**() returns the ACL disk block to the available pool. The routine **c2iread**() reads i-node security information from disk memory.
fs/c2/c2nami.c	The routine **c2namei**() performs the following: • Ensures that the user has permission (**PERMIT_SUIDGID**) to link a setuid or setgid file • Ensures that the security levels of linked files are equal • Ensures that the security level of a new directory is equal to or greater than the security level of its parent directory • Ensures that the user's active security level and compartments are equal to the security level and compartments of the directory to be removed • Ensures that the user's active security level and compartments are equal to the security level and compartments of the file to be removed The routine **c2setattr**() checks the user's active security level and compartments, ensures that the user has permission (**PERMIT_SUIDGID**) to change the mode of a setuid/setgid file, and disallows a change-owner request for a setuid/setgid file when the user does not have the appropriate permission (**PERMIT_SUIDGID**).

Module	Description
fs/c2/c2subr.c	The routine **c2access**() applies discretionary and mandatory access controls. This is the major security module controlling file access.
fs/c2/c2sys3.c	When a secure file system is mounted, the routine **c2mount**() checks to ensure that its security level fits within the security-level range set for the system.
os/chkpnt.c	The routine **chkpnt**() ensures that only a trusted subject process can checkpoint an entire job. It also ensures that the calling process' compartments and security level are a superset of those of the target process set being checkpointed. It gives the restart file the same security level and compartments as the original processes.
os/exit.c	The routine **slgeoj**() makes end-of-job entries in the security log (**/dev/slog**).
os/main.c	The **main**() routine contains instructions to set the system's lower and upper security levels. It also sets the trusted subject user validation information: lower and upper security levels; active security level; active compartments; authorized compartments; and permissions. **main**() also initializes the security log (**/dev/slog**) and the system console.
os/restart.c	The routine **restart**() sets process security level and compartments upon recovery of a checkpointed process.
os/sys2.c	The routine **copen**() forces the mode to **APPEND** for a write-upward open request (a write open request of a file at a security level higher than that of the user who has write-up permission). If a user without write-up permission makes a similar write request, this request fails. **copen**() also checks for super-user and security-administrator privileges when creating a restart file.

Module	Description
	The routine **link**() ensures that the user's active security level and compartments are equal to the file's security level and compartments for link requests.
os/sys4.c	The routine **setthetime**() makes a security log entry.
	The routine **chmod**() ensures that the user has permission (**PERMIT_SUIDGID**) to set a file's mode to setuid/setgid.
	The routine **chdir**() tracks the security path for use in the security log.
	The routine **umask**() ensures that the user has permission (**PERMIT_SUIDGID**) to set the mode mask for a setuid/setgid file.
	The routine **utime**() checks that the user's active security level is equal to the file's security level and that the user's active compartments are a proper subset of the file's compartments before completing the **utime**(2) request.
os/sys5.c	The routine **sysconf**() tests for the secure state of UNICOS (**SC_CRAY_SECURE_SYS**).
os/subr.c	The routines **spath**() and **upath**() contain the security path tracking code; these routines are used if security path tracking is enabled. This code saves the relative path name being referenced in **u->u_vpath** for use by the security log processing routines.

Kernel Stack

This section describes the kernel stack conventions for CRAY-2 systems. It covers the following topics:

- Kernel stack structure
- The **errregs** area

The *current stack frame* is in local memory and the *previous stack frame* is in common memory. Therefore, in a dump of the kernel stack, all stack frames are complete, except for the current stack frame.

When a kernel error exit is processed, information about the current stack frame is copied from local memory to a common memory area at **errregs**. **errregs** is a table defined in **md/error.s**.

Kernel stack structure

The kernel stack consists of a series of stack frames, one for each function on the stack.

Figure 14 shows the structure of a kernel stack frame.

	Value of register S0	**$fp+0 ($fpidnt)**
	Value of register A0	**$fp+1 ($fprtrn)**
Frame package information ($fp)	Previous stack frame base address	**$fp+2 ($fpprev)**
	Stack frame base address	**$fp+3 ($fpbase)**
	Top of stack address	**$fp+4 ($fptop)**
	Segment limit	**$fp+5 ($fpsegl)**
	Caller's FPSIZE + LMSIZE	**$fp+6**
	Function name	$fp+7 **($fpname)**
Saved local memory ($lm)	Saved local memory data for this function (register arguments and variables)	
Auto variables	Nonregister auto variables and compiler temps	Register A2 points here
Arguments	Arguments for this function	Register A6 points here
Top of Stack -->		

Figure 14. Kernel stack frame

All areas except **$fp** are optional; they are included only if necessary.

The frame package information area and the saved local memory area are not valid for the current stack frame, because information for the current stack frame is in local memory at **$fp** and **$lm**, respectively. (All areas are valid for the previous stack frame.) The frame package information area and the saved local memory area for a function are not written to the stack until another function is called. Specifically, the save operation is performed by the **centry** code of the process being called.

The following subsections describe the elements in the kernel stack frame.

Frame package information ($fp)

The fields in the stack frame are described as follows:

Field	Description
$fp+0 ($fpidnt)	Contains the value of register S0 in the following format:

13 bits	19 bits	32 bits
arg_count	*line_number*	*entry_address*

These fields have the following meanings:

arg_count	Number of arguments for the function
line_number	Line number of the beginning of the function
entry_address	Entry-point address of the function

$fp+1 ($fprtrn)	Contains the value of register A0; this is the return address of the calling process.
$fp+2 ($fpprev)	Contains the previous stack frame base address (caller's stack frame base address).
$fp+3 ($fpbase)	Contains the current stack frame base address.
$fp+4 ($fptop)	Contains the top of the stack address. This is the stack frame base address for the next stack frame.
$fp+5 ($fpsegl)	Contains the segment limit (the maximum size of the stack segment)
$fp+6	Calling routine's frame package information area; contains the size of the frame package for the associated function
$fp+7 ($fpname)	Contains the name of the function. Before a return from a function call, **$fp+7** is changed to "--*name*".

Saved local memory ($lm)

This area is allocated in the following order:

1. Register-resident arguments, if any
2. Register-resident auto variables
3. Compiler-allocated scratch registers

Any register arguments or register variables, along with some stack variables (if possible), are stored in local memory.

Auto variables

This area contains sequentially allocated memory locations for the associated function's auto variables that are not register resident. It also contains the compiler's temporary variables.

Arguments

This area contains sequentially allocated memory locations for the arguments to the associated function. Register-resident arguments are also allocated here, but are not used, because they are in local memory. The first argument takes the first word, the second argument takes the second word, and so on.

The errregs area

When the kernel processes an error exit, the first thing it does is save the following items in the common memory table **errregs**:

- Frame package information
- Local memory
- A registers
- S registers
- Two of the vector registers

There is an entry allocated in **errregs** for each CPU. Each entry is 0321 words in length.

Figure 15 shows the format of this area. You can examine **errregs** for all four CPUs by using the **crash**(1M) utility with the command **od errregs 01504**.

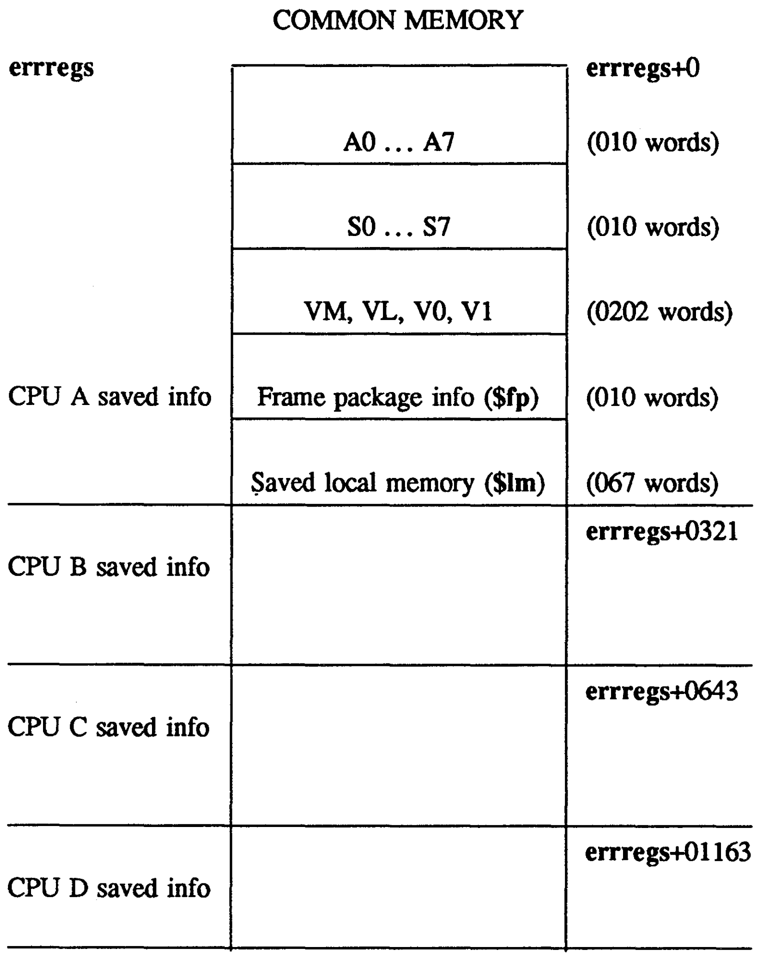

Figure 15. Format of **errregs** area

Trace Buffer

This section describes the trace buffer and provides other details concerning operating system traces.

The trace buffer is a circular buffer that contains entries created with the trace macro **UTRACE**. Information in the trace buffer is used in debugging the system; the **crash**(1M) directive **traceb** prints trace buffer entries.

Format of trace buffer

The trace buffer is initialized (in **md/trace.s**) to all 0's. The symbol **etbegin** marks the first entry in the trace buffer. A "next" pointer (**etindex**) is used to point to the next available slot in the buffer. After a trace entry is written in the buffer, **etindex** is advanced to the next open entry. Buffer entries are reused often on a busy system (on the order of several times every second).

The trace buffer is at address 02000 in common memory.

An entry in the trace buffer is 4 words in length; it has the following format:

string	*CPU, RTclock*	*Parameter_1*	*Parameter_2*

The components of the entry have the following descriptions:

Component	Description
string	Word containing a descriptive string, supplied by the programmer, that uniquely identifies the trace entry. This is the parameter **M** in the **UTRACE** macro call.
CPU	CPU in which the process is running. This value is 7 bits in length.
RTclock	Real-time (RT) clock time. This value is 54 bits in length.
Parameter_1	Word containing the first parameter; this value is something the programmer considered to be useful in debugging. This is the parameter **X** in the **UTRACE** macro call.
Parameter_2	Word containing the second parameter; this value is something the programmer considered to be useful in debugging. This is the parameter **Y** in the **UTRACE** macro call.

The following example shows some example entries in the trace buffer. (This output was generated with the **crash**(1M) directive **traceb** on a CRAY-2 system running UNICOS 5.0.2.)

String	CPU	RT Clock	Parameter 1	Parameter 2
trap	A	7270385104865	122	0636357
sleep	A	7270385141073	0201500003162775	0636357
swtch	A	7270385151791	0636357	0630377
resume	A	7270385152313	0630377	0
resume2	A	7270385152860	02	0237720201
resume3	A	7270385165361	02	0237700200
trap-end	A	7270385178190	0630377	056050000
trap	B	7270385179354	19	0610651

Trace control bitmask

The trace control bitmask, **ostctrl**, is defined in the file **cf/conf.c**. There is also a copy of the mask, called **ostc**, in local memory. Traces are enabled by setting the appropriate bit in the trace mask.

The trace bits **T_CLOCK** and **T_LOCKS** are off by default in the released system. Enabling these traces significantly increases the trace activity.

Note

The trace mask is determined by CRI and is set in the released system. Although generating traces takes resources, this information is considered important. Note that if a site turns off existing traces, CRI may not be able to assist in debugging dumps.

Trace type definitions

The trace type bits correspond with bits in the trace mask. They are defined in the include file **sys/sysmacros.h** as follows:

```
/* ostctrl bits */

#define T_CLOCK   00000000001
#define T_LOCKS   00000000002
#define T_MALLOC  00000000004
#define T_HY      00000000010
#define T_DISK    00000000020
#define T_SYSCALL 00000000040
#define T_RDWR    00000000100
#define T_BIO     00000000200
#define T_SWTCH   00000000400
#define T_SLEEP   00000001000
#define T_IGET    00000002000
```

```
#define T_PROCS   00000004000
#define T_INIT    00000010000
#define T_CONS    00000020000
#define T_EXEC    00000040000 /* also known in inout.s */
#define T_PIP     00000100000
#define T_TTY     00000200000
#define T_NET     00000400000
#define T_SWAP    00001000000
#define T_LSP     00002000000
#define T_TS      00004000000
#define T_CHKPNT  00010000000
#define T_RESTART 00020000000
#define T_SL      00040000000 /* Super Link */
#define T_HSX     00100000000 /* HSX */
#define T_THREAD  00200000000 /* thread */
#define T_ULTRA   00400000000 /* ultra net */
#define T_LSX     01000000000 /* lsx channel */
#define T_FREEZE  02000000000
#define T_CPU     04000000000 /* cpu device */
```

Trace entries in kernel code

Appendix A contains instructions for producing a list of all calls to the trace macro **UTRACE** from current source code.

Appendix Section

Appendix A: Producing a List of Trace Entries

The following procedure outlines how to produce a list of trace macro calls in the kernel source code.

Creating the list

First, create a data file consisting of all the trace macro calls, as follows:

```
cd /usr/src/uts
grep -n UTRACE */*/*.c > infile
grep -n UTRACE */*.c >> infile

cd /usr/include/sys
grep -n UTRACE *.h >> infile
grep -n UTRACE */*.h >> infile
```

This file is sorted in order of modules; the subdirectories are in the order **fs**, **io**, **md**, **os**, **include/sys**.

Sorting the output

You may find it useful to have a list sorted on one of the fields in the trace macro call. The following subsections present several methods to do this.

Sorting by trace type

A list sorted on the trace type can be useful if you are considering turning bits in the trace mask on or off. (See the disclaimer on page 151, however, before changing the trace mask bits.) This enables you to see all the trace macro calls that would be affected by the change. To produce a list sorted by trace type, use the following command:

```
sed  's/^\(.*\)\(T_[A-Z]\)\(.*\)$/\2     \1\2\3/'  infile | sort -df
```

The output from this command appears as follows:

```
T_BIO      io/mdsk.c:158: UTRACE(T_BIO, 'mdskstr0', (long)bp->b_flags<<32|(long)bp,
T_BIO      os/bio.c:593:  UTRACE( T_BIO, 'grabbed',
T_BIO      os/bio.c:600:  UTRACE( T_BIO, 'getblk-s',
T_BIO      os/bio.c:613:  UTRACE( T_BIO, 'gb-dump',
T_BIO      os/bio.c:627:  UTRACE( T_BIO, 'getblk-r',
T_BIO      os/bio.c:655:  UTRACE( T_BIO, 'getblk-b',
T_BIO      os/bio.c:696:  UTRACE( T_BIO, 'getblk', (long)ioh<<32|(uint)bp, (long)dev<<32|
T_BIO      os/bio.c:714:  UTRACE( T_BIO, 'getpblks', &pfreelist, 0 );
T_BIO      os/bio.c:724:  UTRACE( T_BIO, 'getpblk', bp, 0 );
T_BIO      os/bio.c:747:  UTRACE( T_BIO, 'geteblk', bfl, bsize );
T_BIO      os/bio.c:779:  UTRACE( T_BIO, 'geteblk', (long)ioh<<32|(uint)bp, ioh->io_busy);
T_BIO      os/bio.c:794:  UTRACE ( T_BIO, 'brelse', bp, (long)bp->av_forw<<32 | flags);
T_BIO      os/bio.c:936:  UTRACE( T_BIO, 'iomove',(long)bp->b_ioh<<32|(uint)bp,
```

Sorting by string

Perhaps most useful of all is a list sorted on the string in the trace macro call. This string appears in the trace buffer entry and uniquely identifies the trace macro call, so it is useful for debugging. The following command produces a list of these strings in alphabetical order:

```
sed  "s/^\([^']*\)'\([^']*\)'\(.*\)$/\2     \1'\2'\3/"  infile | sort -df
```

The output from this command appears as follows:

```
aread      os/bio.c:152:      UTRACE( T_RDWR, 'aread', ioh, (long)dev<<32|blkno );
ATRACK     io/ddsk.c:408:      UTRACE( T_DISK, 'ATRACK', head, dp->atrack );
awrite     os/bio.c:282:      UTRACE( T_RDWR, 'awrite',(long)ioh<<32, (long)dev<<32|blkno
badaddr    md/machdep.c:307:  UTRACE( T_THREAD, 'savec_th', 'badaddr', 0 );
bdwrite    os/bio.c:396:      UTRACE( T_RDWR, 'bdwrite', (long)bp->b_ioh<<32|(uint)bp,
binit      os/bio.c:980:      UTRACE( T_INIT, 'binit', 0, 0 );
bmxdemop   io/bmxdem.c:51:      UTRACE (T_TS, 'bmxdemop', 0, 0);
bread      os/bio.c:252:      UTRACE( T_RDWR, 'bread', dev, blkno );
brelse     os/bio.c:794:      UTRACE ( T_BIO, 'brelse', bp, (long)bp->av_forw<<32 | flags);
bwrite     os/bio.c:362:      UTRACE( T_RDWR, 'bwrite', (long)bp->b_ioh<<32|(uint)bp,
c2iinit    fs/c2/c2iget.c:483: UTRACE( T_INIT, 'c2iinit', rootdev, swapdev );
c2iread    fs/c2/c2iget.c:48:  UTRACE( T_IGET, 'c2iread', ip, dev );
```

Glossary

Glossary

block	A logical term denoting an arbitrary amount of data; generally a synonym for a disk hardware sector. A block is the smallest allocation unit in a file system. It is the size of one physical disk sector, or 512 words (4096 bytes).
block special file	A special file that reads and writes a block device and buffers its data through the kernel's buffer cache. Its i-node contains the major and minor device number of the logical disk it represents. Block special files appear in **/dev** as follows (when the **ls -l** command is used): `brw-------  1 root   root   0, 0  Aug 25  1988  dsk/root`
boot	The act of starting the system. During an initial system boot (a *slow boot*), the SCC program loads the kernel image from the hard disk on the system console. The term *boot* also refers to the **osboot** program (used for a *fast boot*) and the **boot**(2) system call.
buffer pool	A kernel cache of copies of disk blocks representing blocks on any block device. Typically, these are the disk devices comprising the file systems. The **sync**(2) system call flushes the cache, ensuring that the file systems physically on disk are consistent. Also called a *buffer cache.*
CEA systems	Abbreviation for CRAY X-MP EA and CRAY Y-MP systems.

character special file	A special file that reads and writes a character device and does not buffer data through the kernel's buffer cache. Character special files appear in **/dev** as follows (when the **ls –l** command is used): `crw-w--w- 1 operator operator 1, 0 Apr 3 09:06 console` The i-node for a character special file contains the major and minor device number of the device it represents.
child process	A duplicate of the parent process created with the system call **fork**(2). A child process inherits the environment of the parent process. **SIGCLD** signals death of child to the parent process (see *zombie*).
click	In UNICOS, 512 words. The term *click* implies an area in memory; the term *block* implies an area on disk.
cluster	A file system that consists of one or more physical disk partitions. A file system descriptor file is used to represent a cluster externally; the format of this file is defined by the **fsdesc** structure in the include file **sys/fs/c2fsdesc.h**. A cluster is represented internally in the **c2mount** table.
core file	A file to which an image of a process is written in response to various errors. The core file may be used for post-mortem debugging of the problem. (See **core**(4F) for information on core file format and **signal**(2) for information on when a core file is written.)
CPU	Central processing unit.
current directory	Each process has one directory i-node in the **inode** table designated (by the pointer **uc_cdir**) as its current directory. A process inherits its initial current directory from its parent process. The current directory may be changed with the **chdir**(2) system call. All path names that do not begin with a slash (/) are resolved from the current directory.
CX/1 systems	Abbreviation for CRAY-1 and CRAY X-MP systems.

daemon	Any program that executes continually, waiting to respond to certain events. One example is **errdemon**(1M), which collects error records from **/dev/errlog** and puts them in **/usr/adm/errfile** (by default).
/dev	A directory that normally contains special files. Although special files can exist at any point in the file system, they are usually kept under the **/dev** directory for administrative reasons.
directory	A directory consists of the same components as a file (an i-node and its associated data), except that its data consist of a series of directory entries that are i-node numbers of files or other directories in the file system. A directory contains at least two links (. and ..); these are referred to, by convention, as *dot* and *dot-dot*, respectively. *Dot* refers to the directory itself, and *dot-dot* refers to its parent directory. The format for a file-system-independent directory is defined by the **dirent** structure in the include file **sys/dirent.h.**
dynamic block	A disk block containing information about the current state of the file system. A dynamic block is pointed to by the super block and is located near the center of the partition. The dynamic block is defined by the structure **dblock** in the include file **sys/fs/c2filsys.h**. (See **fs**(4F) for more information.)
effective user ID effective group ID	An active process has an effective user ID and an effective group ID that are used to determine file access permissions. The effective user ID and effective group ID are equal to the real user ID and real group ID of the process, respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID (*setuid* or *suid*) bit or set-group-ID (*setgid* or *sgid*) bit set; see **exec**(2).
file	The data contained in a logically connected series of blocks (viewed by the user as a simple string of bytes). Always identified by an i-node, a file has one or more links in the file system that serve as the terminating names in paths to the file. A file must have at least one link; otherwise it has no name and does not exist because the system deallocates an i-node with zero links.

file access permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

- The effective user ID of the process is super user.
- The effective user ID of the process matches the user ID of the file owner, and the appropriate access bit of the owner portion (0700) of the file mode is set.
- The effective user ID of the process does not match the user ID of the file owner, and the effective group ID of the process matches the group of the file, and the appropriate access bit of the group portion (0070) of the file mode is set.
- The effective user ID of the process does not match the user ID of the file owner, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the other portion (0007) of the file mode is set.

Otherwise, the corresponding permissions are denied.

file descriptor

A small integer used to do I/O processing on a file. The value of a file descriptor ranges from 0 to **NOFILE**−1. (**NOFILE** is defined in the include file **sys/param.h**.) A process may have no more than **NOFILE** file descriptors open simultaneously. A file descriptor is returned by system calls such as **open**(2) or **pipe**(2). Calls such as **read**(2), **write**(2), **ioctl**(2), and **close**(2) use the file descriptor as an argument.

The kernel uses the file descriptor as an index into the per-user array of open file pointers (the **uc_ofile** field in the user common (**ucomm**) area); each entry in this array points to an entry in the file table, which, in turn, points to an entry in the **inode** table, which contains the i-node.

file name

You may use names consisting of 1 to 14 characters (**DIRSIZ**, defined in **sys/param.h**) to name an ordinary file, special file, or directory. Select these characters from the set of all character values excluding \0 (null) and the ASCII code for / (slash). Using "*", "?", "[", or "]" as part of a file name is generally unwise because of the special meaning attached to these characters by the shell (see **sh**(1)). Although permitted, using unprintable characters in file names is also inadvisable.

file pointer

The pointer (**f_offset**) defining the position at which reads and writes are to start in an open file. The position is updated by the **read**(2) or **write**(2) system call, based on the number of bytes transferred. The pointer can also be set with the **lseek**(2) system call.

	The file pointer is contained in the system file table, which is defined in the include file **sys/file.h**.
file system	A tree-structured collection of files and their associated data and attributes.
FSS	File system switch.
GID	Group identification number.
header file	C language files that contain definitions used by more than one program (such as compilers, assemblers, and system utilities), often for data interchange between programs. System header files are called *include files*; they are found in the directory **/usr/include/sys**.
home directory	The main directory of a user's account; the user's home directory becomes the current directory of the user's shell when the user first logs in to the system. The home directory is stored in the user common (**ucomm**) area as an i-node pointer.
include file	See "header file."
i-node	A file identifier, containing the file's state information and pointers to the data composing the contents of the file. There are two versions of every i-node, the disk i-node and, when the i-node is in use, an in-core i-node. The disk i-node may have a different format for each type of file system supported; the disk i-node for the C2FS file system is defined by the **dinode** structure in the include file **sys/ino.h**. The format of an in-core i-node is independent of the file system type; it is defined by the **inode** structure in the include file **sys/inode.h**.

	The C2FS-dependent i-node is defined by the **c2inode** structure in the include file **fs/sys/c2inode.h.**
	This term is written as "inode" in some source modules.
job ID	Each active process may be a member of a job that is identified by a positive integer called the job ID (*JID*). This grouping allows limiting of resource usage (that is, CPU and memory) for a set of related processes.
kernel mode	The state a CPU is in when executing the kernel. Every CPU is always executing on behalf of some user, either in user mode (executing user code) or in kernel mode.
large file	A file larger than 8 sectors, or a file created with **O_BIG** bit specified in **open**(2).
link	A directory entry containing a file name and an i-node number; this entry provides a mapping between the path name of the directory entry and the file. A file may have more than one link. A link may not map to an i-node on a different file system.
major clock tick	An interrupt that occurs once every second. Each major tick causes the kernel to adjust the CPU scheduling priority of every process.
major device number	Each regular and special file has a major device number; the kernel uses this number to identify the device driver used when accessing the device.
	Major and minor device numbers for a special file are displayed when the command **ls –l** is used on a special file.
MCC	The Maintenance Control Console (MCC) program; it is used for system maintenance and debugging.

member	A partition on a physical disk that is part of a striped group.
migrated file	A file that has been moved, or archived, to a tape drive. Migrated files appear as follows (when the **ls –l** command is used):

```
mrw-------  1 jhb    opsys    2, 0  Feb  14  1989  myfile
```

The permissions are the same as those for the file before it was migrated.

minor clock tick	A software-generated clock interrupt that occurs once every 1/100 of a second. At each minor tick, each connected process is lowered in priority according to CPU usage, and the run queue is checked for the lowest-priority process to connect.
minor device number	Identifies a specific device in a major device group. Each special file has a minor device number that is passed to the appropriate device driver for the special file. The driver may use the value of the minor device number as it chooses (typically as an index into a list of logical devices).

Major and minor device numbers for a special file are displayed when the command **ls –l** is used on a special file.

mode bits	The bits in an i-node that identify the access permissions for the file and the file type (for instance, block special, character special, or directory).

The following table shows the mode letters and octal permissions (the latter are defined in **sys/inode.h**):

Mode Letter	Octal Permission	Description
	0170000	File type as follows:
p	0010000	FIFO special file (named pipe)
c	0020000	Character special file
d	0040000	Directory
b	0060000	Block special file
–	0100000	Regular file
m	0120000	Off-line file without data
s	0140000	UNIX domain socket (TCP/IP only)
s or S	0004000	Sets UID on execution
s or S	00020*n*0	Sets GID on execution if *n* is 7, 5, 3, or 1
l	00020*n*0	Enables mandatory file/record locking, if the file is a regular file and *n* is 6, 4, 2, or 0
	0000777	Access permissions as follows:
r	0000400	Allows read operations by file owner
w	0000200	Allows write operations by file owner
x	0000100	Allows execute operations (or search operations if a directory) by file owner
---rwx---	0000070	Allows read, write, and execute operations (or searches) by group
------rwx	0000007	Allows read, write, and execute operations (or searches) by others

mount — Properly-formatted file systems may be mounted on a directory. Subsequent references to the directory are mapped to the root of the newly mounted file system.

The mount table, which is defined by the **mount** structure in the include file **sys/mount.h**, is the bridge between the two in-core i-nodes.

multitasking group — A group of processes created via the **tfork**(2) system call; these processes work together in one address space to achieve parallel processing. Also called an *m-group.*

multiuser mode — Any run level intended for time-shared use by many users. Multiuser mode is traditionally associated with run-level 2. For more information, see **inittab**(4F) and the *UNICOS System Administrator's Guide for CRAY-2 Computer Systems*, publication SG-2019.

NFS — Network File System.

nice value	A process's nice value (the value in the **p_nice** field of the **proc** structure) is used in evaluating its priority; the nicer the process, the slower it runs. The nice value is changed with the **nice**(2) system call.
node	A synonym for special file.
NQS	The Network Queueing System, which is a batch queueing subsystem running under UNICOS.
orphan process	A child process whose parent has exited before it has. The **init** process inherits orphan processes.
parent process ID	A new process is created by a currently active process; see **fork**(2). The parent process ID of a process is the process ID of its creator. The parent process ID is often referred to as the *ppid*.
partition	A contiguous set of blocks on a logical device. In file allocation, partitions permit the distribution of files across the physical devices of a file system.
path name	A sequence of branches in the file system tree that defines a route for accessing a file. A path can start at the root or at the current directory. The last segment of the path is one name for the file. A full path name is one that starts with a slash character. A relative path name is one that does not start with a slash character; it is interpreted as starting in the current directory.
permission bits	The bits in an i-node that control access to the file; a subset of the i-node's *mode bits*. See "mode bits."

priorities	The "rank" of a process in terms of execution. If a process' priority (stored in the field **p_pri**) is less than **PUSER**, the priority is the level at which a process sleeps. If a process' priority is greater than **PZERO**, the sleep is interruptible; if the priority is less than or equal to **PZERO**, the sleep is not interruptible. Sleep priorities signify the event for which a process is waiting. The important system priorities are defined in **sys/param.h**. Priorities greater than **PZERO** determine the order of execution for running processes.
process group ID	Each active process is a member of a process group that is identified by a positive integer called the process group ID. (Generally, a process group is the set of all processes controlled by a single terminal; hence, the group leader is typically a user's login shell.) This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see **kill**(2). The process group ID is also referred to as the *pgid*; the field **pc_pgrp** field in the process common (**pcomm**) area contains this value.
process ID	Each active process in the system is uniquely identified by a positive integer called a process ID. The symbol **MAXPID**, which is defined in the include file **sys/param.h**, defines this range as from 0 to 99999. The process ID is also referred to as the *pid*.
pty	Pseudo tty; a logical device that provides a path from networking software to the processes associated with a user's interactive session. The pty has two *sides*: a *control*, or *master*, side and a *slave*, or *subordinate*, side. The user process in this connection is called the *slave side* of the pty. Each user (for example, the **telnetd** daemon and a shell) opens a different major device type but the same minor device type, so each is sharing the same buffers but addressing a driver that reads and writes the buffer as a mirror image of the other. The control side typically communicates with a front-end terminal over a TCP/IP socket or the HYPERchannel. The special files in **/dev** that represent the pty pseudo-devices are named, by convention, **/dev/pty/***nnn* and **/dev/ttyp***nnn*.

real user ID real group ID simultaneous group ID	Each user allowed on the system is identified by a positive integer called a *real user ID*. Each user is also a member of a group. The group is identified by a positive integer called the real group ID. An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process. Each user can belong to multiple groups simultaneously. The largest number of groups to which a user can belong is **NGROUPS** (defined as 64 in **sys/param.h**).
root	This word generally has four distinct meanings, depending on context: • The primary file system that is always mounted; the root file system. • The traditional account name associated with UID 0. This UID is called the root or super user. • The first level of any file system; the root of a file system. • The partition, in a cluster, containing the root directory of the entire file system.
root directory	Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system. The fields **uc_rdir** and **uc_cdir** in the user common (**ucomm**) area point to the root directory and current working directory, respectively, for the user. See also "home directory."
run level	A system software configuration controlled by the contents of the **/etc/inittab** file (see **inittab**(4F)) and the **init** process (**proc[1]**).
SCC	The system control console program, used during all normal system operation.
script	A text file consisting of commands to be executed by a shell program. Also called a *shell script*.

sector	A 4096-byte unit of disk space (see *block*).
sector allocation	Method of allocation used for sector files.
secure system	A system built with the UNICOS security feature enabled; that is, with the **SECURE_SYS** and **SECURE_INC** parameters in the include file **sys/param.h** set to 1. This directs the compiler to compile the security code bounded by the directives **#if SECURE_SYS** and **#endif SECURE_SYS**.
setgid	A specific bit in the mode bits of an i-node, indicating that any user who executes the program contained in the corresponding file assumes the effective GID of the file's group.
setuid	A specific bit in the mode bits of an i-node, indicating that any user who executes the program contained in the corresponding file assumes the effective UID of the file's owner.
Share share	The term *Share* refers to the fair-share scheduler. The term *share* refers to a unit of allocated resources.
shell	Any command interpreter. In addition to interpreting and executing user commands, a shell generally provides various programming constructs (for example, variable assignment, expression evaluation, and conditional execution) for greater utility in writing scripts. UNICOS supports two traditional UNIX shells, **sh**(1) (the Bourne shell) and **csh**(1) (the Berkeley-derived C-shell).
single-user mode	A run level intended for use by a single user working from the system console.
small file	A file 8 sectors or smaller (see "large file").

special files	Directory entries that provide access to peripheral devices and other system resources. Special files do not contain data; rather, they provide the kernel with the information (namely, major and minor device numbers) needed to access specific devices. There are two types of special files: character special files and block special files. Protection of and access to files is the same for both special and regular files. Access to logical as well as physical devices is possible through this mechanism. For example, access to main memory and a communication path for logging hardware errors are available. Special files are typically stored in the **/dev** directory. A special file is also known as a *node* or *device node*.
special processes	The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as **proc[0]** and **proc[1]**. **proc[0]** is the memory scheduler, or *swapper*, and **proc[1]** is the initialization process (**init**()). **proc[1]** is the ancestor of every other process in the system and controls the process structure.
striping	Writing to multiple disk devices as a single group, with data blocks interleaved among the members for maximum throughput at very high bandwidth. The set of disk devices is referred to as a *striped group*.
super block	A disk block containing configuration information for a given file system. The super block is replicated across all blocks of the first track of each partition. The super block is defined by the structure **sblock** in the include file **sys/fs/c2filsys.h**.
super user	Any user (and by extension, process) whose effective user ID (UID) is 0. UID 0 may override the normal UNICOS permission mechanism.
system mode	See "kernel mode."

TCP/IP	Transmission Control Protocol/Internet Protocol. TCP/IP is the set of software that enables Cray computer systems to communicate with other systems that also use these protocols. Much of TCP/IP is implemented in the kernel.
trusted subject	Term used with the UNICOS security feature; a process or daemon that has the authorized compartments **secadm**, **sysadm**, **sysops**, and **unicos**. Trusted subject processes include **main**, **sched**, **init**, **getty**, and **login**. Trusted subject daemons include the NQS daemon (**nqsdaemon**(1M)), the USCP daemon, and the tape daemon (**tpdaemon**(1M)).
tty	Commonly used as a synonym for **terminal**. A *terminal* can be a physical terminal, or it can be the slave side of a pseudo terminal (or *pty*). The special files (nodes) in **/dev** that are named, by convention, **/dev/ttyp***nnn* represent the slave side of the pseudo terminals. The special file **/dev/tty** is the control terminal associated with a process group.
tty group ID	Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group; see **exit**(2) and **signal**(2). The tty group ID is the same as the process group ID (pgid) that is stored in the **pc_pgrp** field of the process common (**pcomm**) area.
UID	User identification number.
/unicos	The file name of the currently executing kernel binary image. Various commands, such as **crash**(1M) and **ps**(1), use this file to access the kernel symbol tables.
USCP	The UNICOS Station Call Processor; a daemon that runs under UNICOS and allows users on front-end systems running CRI link (station) software access to the system through the COS SCP interface.

User database (UDB)	CRI's enhancement of the traditional **/etc/passwd** and **/etc/group** files.
user mode	The state a CPU is in when executing user code. Every CPU is always executing on behalf of some user, either in user mode or in kernel mode (executing kernel code).
utsname	A structure that is the list of values that identifies the system to programs. The first field is **sysname**; this is the name of the current version of the system. The **utsname** is available by using the **uname**(2) system call.
zombie process	A process that has exited and, optionally, sent a **SIGCLD** signal to its parent. The process retains its process table entry until its parent process executes a **wait**(2) system call to collect the status of the exited child process. Zombie processes are marked with a "Z" in the output of **ps**(1), as shown in the following example:

```
F  S  UID   PID   PPID  CM  PRI  NI  ADDR   SIZE  WCHAN   TTY  TIME  COMMAND
1  S  1082  3417  1     0   28   20  64022  209   605415  004  0:00  ksh
1  Z  1082  3462  3417  0   990  24                            0:00  <defunct>
```

This fragment of **ps** output shows the status of a zombie process and its parent process. (The zombie was executed as a background process.) The parent is waiting for terminal I/O (PRI is equal to **PPTY**); it will likely perform the **wait**(2) operation after the user types the next command.

READER'S COMMENT FORM

UNICOS Internal Reference Manual for CRAY-2 Computer Systems SP-2023 5.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
   ____ other (please specify): ______________________
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
   ____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____	8) Physical qualities (binding, printing) ____
6) Completeness ____	9) Readability ____
7) Organization ____	10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name ______________________	Address ______________________
Title ______________________	City ______________________
Company ______________________	State/ Country ______________________
Telephone ______________________	Zip Code ______________________
Today's Date ______________________	

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

READER'S COMMENT FORM

UNICOS Internal Reference Manual for CRAY-2 Computer Systems SP-2023 5.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
   ____ other (please specify): ____________________________
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
   ____for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name ______________________
Title ______________________
Company ______________________
Telephone ______________________
Today's Date ______________________

Address ______________________
City ______________________
State/ Country ______________________
Zip Code ______________________

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120