Cray Research, Inc. Proprietary

Date November 1995
Revision K
Release 8.0

UNICOS Internals Technical Reference

TR-ITR 8.0 K Volume I

This document is intended for instructional purposes. It should not be used in lieu of other Cray Research, Inc. publications.

This document may describe a hardware or software product that has not been officially released. Existence of any such description is not a commitment of the actual release or support by Cray Research, Inc.

Software Training

Cray Research, Inc.

CRAY RESEARCH PROPRIETARY

Dissemination of this documentation to non-CRI personnel requires approval from the appropriate vice president and a nondisclosure agreement. Export of technical information in this category may require a Letter of Assurance.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the subparagraph [(c) (1) (ii)] of the rights in Technical Data and Computer Software clause at 52.227–7013. (October 1988)

Cray Research, Inc.
655 Lone Oak Drive
Eagan, MN 55121

Cray Research, Inc. Unpublished Private Information – All Rights Reserved.

Autotasking, CF77, CRAY, Cray Ada, CRAY Y-MP, CRAY-1, HSX, SSD, UniChem, UNICOS, and X-MP EA are federally registered trademarks and CCI, CF90, CFT, CFT2, CFT77, COS, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, Cray NQS, Cray/REELlibrarian, CraySoft, CRAY T90, CRAY T3D, CRAY X-MP, CRAY XMS, CRAY-2, CRInform, CRI/*Turbo*Kiva, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, HEXAR, IOS, LibSci, Luminary, MPP Apprentice, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNET, RQS, SEGLDR, SMARTE, SUPERCLUSTER, SUPERLINK, Trusted UNICOS, and UNICOS MAX are trademarks of Cray Research, Inc.

DECnet is a trademark of Digital Equipment Corporation. HYPERchannel and NSC are trademarks of Network Systems Corporation. Kerberos is a trademark of Massachusetts Instititute of Technology. LANlord is a trademark of Computer Network Technology Corporation. NFS, ONC, Sun, and Sun Workstation are trademarks and RPC is a product of Sun Microsystems, Inc. UltraNet is a trademark of Ultra Network Technologies, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. X Window System is a trademark of X Consortium, Inc.

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

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk (612) 683–5907
Fax number (612) 452–0141

Record of Revision

This document is designed for use in Training and supports the software release version identified below. Some of the information and related examples may not be consistent with the software release version currently running at your site.

Version	**Description**
G	1992-1993, UNICOS 7.0
H	1993, Enhancements and modifications relating to 7.0 kernel changes
I	December 1993, Enhancements and modifications relating to 7.0/7.C changes
J	December 1994, UNICOS 8.0
K	November 1995, UNICOS 8.0, revisions and addition of disk drivers

This page is blank intentionally.

Contents

Preface iii
Conventions iii
Man page references iv
Ordering publications v
Reader comments v

Software Overview [1] 1–1
Objectives 1–1
Overview 1–2
Recommended reading 1–3
Recommended outside reading 1–4
UNIX bibliography 1–4
System functional overview 1–6
Session from a real terminal 1–6
Session from a pseudo terminal 1–8
Interactive session from a Cray station 1–10
Batch access from a Cray station 1–12
Kernel organization overview 1–14
System memory organization 1–14
Source code organization 1–16
System-wide files 1–16
UNIX time-sharing system `uts/` (kernel source code) 1–17
Object code organization 1–20
Kernel logical organization 1–22
Kernel structures for process control 1–26

Hardware Overview [2] 2–1
Objectives 2–1
Hardware Identification by serial number 2–2
CRAY Y-MP System with IOS model D 2–4
CRAY Y-MP System with IOS model E 2–6
IOS model E 2–8
I/O overview 2–8
General packet format 2–10
Packet types 2–10
Disk request packet 2–12
Simple disk read 2–12
Write behind 2–14
No write behind 2–14
CRAY Y-MP EL differences summary 2–16
CRAY Y-MP system control 2–19
Registers in each CPU 2–20
Shared resources 2–30
CRAY Y-MP C90 system control 2–32
Registers in each CPU 2–32
Status registers 2–34
Shared resources 2–34

CRAY X-MP system control 2–35
Registers in each CPU 2–35
Shared resources 2–35
Interrupt processing summary 2–36
Exchange (XP information) 2–40
Exchange package `crash` display 2–40
XP mode and status bit breakdown 2–40
Memory addressing modes 2–42
CRAY-1 and CRAY X-MP without EMA 2–42
CRAY X-MP in compatibility modes 2–44
CRAY X-MP EMA instruction formats 2–47
CRAY Y-MP compatibility (24-bit) mode 2–49
CRAY Y-MP EAM (32-bit) mode 2–51
CRAY Y-MP EAM instruction formats 2–53
Summary of hardware types / binary restrictions 2–55
CRAY Y-MP C90 in native mode 2–57
CRAY Y-MP C90 in compatibility mode 2–57
CRAY X-MP in compatibility mode 2–57
Hardware system control foldouts 2–59
CRAY Y-MP system control 2–61
CRAY Y-MP C90 system control 2–62
CRAY X-MP system control 2–63

System Initialization [3] 3–1
Objectives 3–1
Overview 3–2
Kernel compile options 3–3
Kernel code optimization 3–3
Global register assignment 3–3
Global intrinsic functions 3–4
Vector use restrictions 3–5
Kernel mode intrinsic functions 3–5
Kernel mode intrinsic functions for CRAY Y-MP C90 systems 3–6
Assembler table macros 3–7
UNICOS linked lists 3–8
UNICOS kernel bit maps 3–10
UNICOS stacks 3–12
Stable stack feature summary 3–13
Stack pool management 3–14
`stackinit()` 3–14
`expandstack()` 3–14
`contractstack()` 3–14
Stack management 3–15
`allocstack()` 3–15
`freestack()` 3–15
Stack format 3–16
Context switching 3–18
CPU and process management 3–18
Basic principles 3–18
Kernel register save areas 3–18

Context switch sample 3–20
sleep() and wakeup() 3–22
Kernel main loop overview 3–24
Kernel multithreading 3–26
Overview 3–26
Lock mechanics 3–28
UNICOS multithread lock logic - general 3–29
SEMLOCK macro 3–30
SEMLOCK illustration 3–31
MEMLOCK macro 3–32
MEMLOCK illustration 3–33
ATOMIC lock macros 3–34
ATOMIC_ADD illustration 3–35
R_MEMLOCK and W_MEMLOCK lock macros 3–36
R_MEMLOCK and W_MEMLOCK illustration 3–39
Atomic sleep 3–40
Logic without atomic sleep 3–41
Logic with atomic sleep 3–41
Ownership macros 3–42
Lock hierarchy 3–42
Lock statistics 3–43
Lock debugging 3–43
Kernel register uses 3–44
Kernel CPU register usage 3–44
Kernel cluster (1) register usage 3–44
Kernel cluster (1) semaphore register usage 3–45
Bootstrapping the mainframe 3–47
Booting methods 3–47
Bootstrapping the mainframe with the full kernel 3–48
Kernel structures at deadstart 3–50
Kernel structures at deadstart 3–51
Bootstrapping the mainframe with a compressed kernel 3–52
UNICOS kernel startup 3–55
Startup overview 3–55
mfstart / mfinit logic 3–56
csl processing 3–58
Startup file / table relocation 3–62
umain() logic 3–67
sysproc() routine 3–73
Summary 3–73
Creating system processes 3–73
sysproc() example 3–74
Creating system processes 3–76
Central memory sizes 3–82

Kernel Mainline [4] 4–1
Objectives 4–1
Kernel mainline overview and mainline detail diagram 4–2
Mainline outer loop 4–8

Kernel entry 4–8
Initial saves and updates 4–8
immtrap - Trapping monitor mode interrupts 4–14
rpescrub and immrpe 4–16
Interrupt handler selection 4–18
Idle processes 4–22
Idle process selection 4–22
Idle process - idle CPU 4–23
Idle process - down CPU 4–24
giveup() and idler 4–26
Process selection 4–28
Signals 4–30
Signal detection 4–30
issig() - Kernel's test for a processable signal 4–32
Catching a signal 4–34
Signal data structures 4–37
Library routines words 4–40
Kernel signal processing overview 4–42
Library signal processing overview 4–45
Kernel exit 4–47
Mainline inner loop - Interrupt handlers 4–51
usrnex - User normal exit (system call) 4–51
System entry table 4–53
usrioi - I/O interrupt 4–60
LOWSP channels 4–62
User error interrupts 4–64
usrfpi - User floating-point interrupt 4–64
usrore - User operand range error 4–64
usrpre - User program range interrupt 4–64
usrbpi - User breakpoint interrupt (C90 only) 4–64
usreex - User error exit 4–64
usrdli - User deadlock interrupt 4–66
usrdli logic 4–68
usrpci - User programmable clock interrupt 4–70
usrmei - User memory error interrupt 4–72
usrmcu - User maintenance control unit 4–76
usrrtm - User real-time interrupt 4–76
usripi - User interprocessor interrupt 4–78
usrrpe - User register parity errors 4–80
usrmii - User monitor mode instruction interrupt 4–82

Process Management Subsystem [5] 5–1
Objectives 5–1
Process management tables 5–2
Regular process - System mode 5–2
Regular process - User mode 5–4
Shared text process - User mode 5–6
Multitasked group - System/user mode 5–8
Proc tables chains 5–10
allproc 5–10

availproc 5–10
allmtask 5–10
idlemtask 5–10
pidhash 5–11
runq 5–11
hsque (sleep queues) 5–12
swapq 5–12
p_children 5–12
pc_sibling 5–12
p_mlink 5–13
sess 5–13
pgrp 5–13
Life cycle of a proc table entry 5–14
Shell command execution 5–14
Process creation 5–16
fork/tfork 5–16
dofork logic 5–16
newproc logic 5–18
fork data structures 5–20
tfork data structures 5–22
User program initiation (exece) 5–24
User commands and shell exece request 5–24
Kernel sets up for the request handler 5–26
Request handler finds the a.out file 5–28
exece 5–28
Store arguments and environment in kernel's exec area 5–30
Load a nonshared executable file's text 5–32
Load a (split text/data) executable file 5–34
Move caller's arguments and environment to the new program 5–36
Finish kernel handler and return to new user program 5–38
Entry to user startup code ($START) 5–40
Entry to and exit from user's main 5–42
vfork() logic summary 5–44
Process control 5–48
Process priorities 5–48
sleep 5–50
wakeup 5–51
swtch 5–52
getstack 5–54
resume 5–55
clongjmp 5–56
sleep examples 5–58
Process termination 5–64
wait logic 5–64
exit processing 5–66
exit calls 5–66
exit logic 5–66
The exit of a younger sibling 5–74
checkpoint/restart 5–76
Summary 5–76

Additional materials 5–76
Structures 5–76
Checkpoint file format 5–77
Buffer management 5–77
Buffer management logic flow 5–77
Checkpoint logic flow 5–78
Restart logic flow 5–80
CPU management 5–82
callout/callarr tables 5–82
callout table organization 5–82
callarr table organization 5–82
Adding callout table entries 5–82
Processing callout table entries 5–82
Cancelling callout table entries 5–82
Interlock mechanisms 5–84
Accessing the callout table via crash(8) 5–84
Managing timed events 5–86
Hardware programmable clock 5–86
User profiling 5–88
Setting up the clock 5–88
At the PCI interrupt 5–88
Resetting the clock 5–88
Real-time scheduling 5–90
Becoming a real-time process 5–90
Declaring real-time compute needs 5–90
At programmable clock interrupts (when user requested CPU_RTFRAME) 5–92
Alarm clock events 5–94
alarm(2) system call 5–94
ioctl(2) system call against /dev/cpu/any 5–94
CPU scheduling 5–96
The clock() function 5–96
CPU scheduling clock intervals 5–98
The minor cycle or "tick" (1/60 sec.) 5–98
The major cycle (1 sec.) 5–98
The share cycle (4 sec.) 5–98
Fair-share scheduler components 5–100
User fair share specifications 5–100
Login time 5–100
Kernel's role 5–100
shrdaemon 5–100
Tuning 5–100
Monitoring 5–100
Testing/debugging 5–100
Fair-share group concept 5–102
clock() routine's role in CPU scheduling 5–104
Nice charge and decay rate tables 5–106
Process scheduling (UNIX versus UNICOS) 5–108
Process scheduling on UNIX systems 5–108
Process scheduling on UNICOS systems 5–108

Fair-share logic 5–110
clock() / share() fair-share logic (minor cycle) 5–110
clock() / share() fair-share logic (major cycle) 5–112
clock() / share() fair-share logic (share cycle) 5–114
Fair-share graphs 5–116
Share off – varying nice 5–116
Share on – varying number per user 5–118
Share on – varying nice 5–120
Share on – varying shares 5–122
Share on – sleep/wakeup 5–124
Share on – general 5–126
Fair-share scheduling parameters – shconsts 5–128
Inodes and limits system call 5–130
Monitoring the fair-share scheduler 5–132
shrates 5–132
shrinfo 5–134
shrmon 5–136
shrstats 5–138
shrusage 5–140
shrview 5–142
User's fair-share information 5–144
Memory management 5–146
Introduction 5–146
Memory mapping 5–148
Memory block management 5–150
Swap device 5–152
Swap space allocation 5–154
sched category 5–155
Shared text 5–156
Overview 5–158
Multiplexed scheduler 5–159
Swap-in queue 5–160
Swap priority principles 5–161
Swap priority factors 5–162
Swap priority calculation 5–164
schedv_init() 5–165
Guaranteed residency time 5–166
nschedv(8) 5–167
Significant variables and functions 5–170
sched() logic overview 5–181
Resource limits 5–197
CPU limits 5–197
Memory limits 5–199
Process limits 5–200
Secondary data segment limits 5–201
Core limit 5–202
File system block limits 5–203
File system quotas 5–204
Cooperative parallel interface – Overview 5–207
Structures 5–208

Logic components 5–210
System components 5–215
resume() 5–215
clock() 5–218
uresch() 5–220
thread() 5–220
Idle CPUs 5–222
System call 212 5–224
User and library components 5–226
Program startup 5–226
Data structures 5–228
User task scheduling - Logic flow 5–230
User task scheduling - Data structures 5–234
Library scheduler's context ready masks 5–236

UNICOS I/O Overview [6] 6–1
Objectives 6–1
Device types 6–2
Block devices 6–2
Character devices 6–2
Device configuration 6–4
Configuration and parameter files 6–4
Device special inode sample list 6–5
File types 6–6
Raw and buffered I/O 6–8
Block I/O methods 6–10
Logical device cache (ldcache) 6–10
Sample buffered read using SSD ldcache 6–10
Sample raw write using BMR ldcache (IOS model B, C, and D only) 6–10
Synchronous and asynchronous I/O 6–12
Synchronous I/O 6–12
Write integrity 6–12
Asynchronous I/O 6–13
listio 6–13
Internal processing 6–13
I/O layers 6–14
Overview 6–14
User process 6–16
File system and file 6–18
Device drivers 6–20
IOS and interrupt handlers 6–22
Vnode VOP operations table 6–24
File system macros 6–25
8.0 versus 7.0 6–25
8.0 file system routines 6–27
8.0 vnode VOP routines 6–27
Vnode/Vfs Operations Tables – UNICOS 8.0 comparison chart 6–29
vop operations 6–29
vfs operations 6–30

Secondary vnode operations table UNICOS 8.0 DFS 6–31
vn operations 6–31
DFS 8.0 Vnode VOPX routines 6–32
Vnode VOPX operations table 6–34
Virtual file system switch table 6–36
Condensed virtual file system switch table definitions `uts/c1/cf/devsw.c` 6–36
Virtual file system switch table definition with CRINFS & DCE 6–36
Block device switch table 6–38
(`bdevsw`) controls access to I/O drivers for block device types 6–38
Block device switch table 6–39
Block device driver definitions (edited) `uts/c1/cf/devsw.c` 6–40
Character device switch table 6–42
Character device driver definitions (edited) `uts/c1/cf/devsw.c` 6–44

File System Management [7] 7–1
Objectives 7–1
File system 7–2
User view 7–2
Internal view 7–4
Internal in-core structural view 7–6
Path names 7–20
Logical disks 7–22
Definition 7–22
Logical disk configuration – Model B/C/D 7–22
Configuration via the startup parameter file 7–22
Startup `param` file 7–22
`pscan()` 7–22
IOS-Param 7–22
Physical device and slice configuration 7–24
Special slice names 7–24
Logical device definitions 7–28
System device definitions 7–30
Configuration tables 7–32
Logical disk addressing 7–32
Logical disks 7–34
Reconfiguration via `mknod`(8) 7–36
Logical disk configuration – Model E 7–38
Configuration via the startup parameter file 7–38
Param file sections 7–38
Reconfiguration via `mknod`(8) 7–40
NC1 file system – Layout 7–44
Partitions 7–44
Partition format 7–46
Superblocks 7–48
Dynamic blocks 7–50
Block allocation bit map 7–52
Inode regions and inode number 7–54
Inode regions 7–54

Inode number 7–54
Partition data blocks 7–56
Partition control fields 7–58
NC1 file system – Example 7–60
Device specifics 7–60
File system specifics 7–62
Example – Octal dump 7–64
NC1 file system – inode description 7–74
Partition layout 7–74
File block addressing 7–76
Directory description 7–78
File format 7–78
Hierarchy 7–80
NC1 file system – Example 7–82
File system specifics 7–82
Octal dump 7–91
File system functions 7–113
Introduction 7–113
Path name processing 7–114
lookup overview 7–114
Path name structure 7–116
Path name operations 7–118
lookup logic flow 7–120
lookupname logic flow 7–121
traverse() logic flow 7–122
lookuppn() logic flow 7–126
Directory name lookup cache 7–128
ncache structure 7–129
DNLC – Structures and routines 7–130
dnlc_lookup() logic flow 7–132
Directory operations 7–134
Directory functions 7–137
Create a directory entry 7–137
Remove a directory entry 7–138
More lookup logic flow 7–139
lookuppn() logic flow 7–141
nc1lookup() logic flow 7–144
VFS and VOP macros 7–145
Inode access (tables) 7–148
nc1iget/nc1iput 7–150
Mount 7–152
Mount of root file system 7–152
mount(2) system call 7–154
Mount onto root file system 7–156
Traversing into a mounted file system 7–158
Traversing out of a mounted file system 7–160
Hard and soft links 7–162
Multilevel symbolic links 7–165
/proc file system 7–166
Introduction 7–166

Process file regions 7–167
Configuration 7–168
mount(2) system call 7–169
open(2) system call 7–170
ioctl(2) debugging features 7–172
lseek(2) system call 7–173
read(2) system call 7–174
Network file system 7–176
Overview 7–176
UNICOS as nfs client 7–176
UNICOS as nfs server 7–178
NC1-related routine locations 7–180

File Management [8] 8–1
Objectives 8–1
open(2) system call 8–2
System file table flags 8–7
Vnode and inode table types, modes, and flags 8–8
File security 8–13
open(2) security 8–13
File access (nc1access()) 8–15
I/O table summary 8–22
File locks 8–24
File lock structures 8–24
Lock settings 8–26
Checking file locks 8–28
Lock conflict resolution 8–29
Allocation and block mapping 8–32
File system allocation and partitions 8–32
File creation/allocation 8–32
Inode allocation 8–33
Inode allocation – open() 8–33
Initial allocations 8–33
Ongoing allocation 8–33
Allocation option RR1STDIR 8–35
Allocation option RRALLDIR 8–36
Allocation option RRFILE 8–37
nc1ialloc() logic 8–38
nc1iassign() 8–40
nc1addinodes() 8–41
Block mapping 8–42
Description 8–42
Block mapping examples 8–44
File data allocation 8–46
nc1alloc() 8–46
write() calling nc1alloc() 8–47
First write (extension) to file 8–48
Subsequent writes (extensions) to file 8–49
Summary - Primary partitions only 8–50

Examples 8–51
Summary - Primary and secondary partitions 8–52
Examples 8–53
Logic 8–54
Read and write 8–57
Read overview 8–57
Interrupt handler (md) layer 8–57
umain 8–57
sdsreq 8–57
System call (os) layer 8–57
read() 8–57
rdwr() 8–57
reada() 8–58
rdwra() 8–58
listio() 8–58
ssread() 8–58
ssrd 8–58
soreceive() 8–58
File system (fs) layer 8–59
nc1read() 8–59
nc1pread() 8–60
Physical (raw) (os bio) layer 8–60
nc1rawio() 8–60
physio() 8–60
aphysio() 8–60
Buffered (os bio) layer 8–61
bread() 8–61
aread() 8–61
getblk() 8–62
Write overview 8–63
Interrupt handler (md) layer 8–63
umain 8–63
System call (os) layer 8–63
write() 8–63
rdwr() 8–63
writea() 8–63
rdwra() 8–63
listio() 8–63
sswrite() 8–63
sswr 8–63
sosend 8–64
File system (fs) layer 8–64
nc1write() 8–64
nc1pwrite() 8–65
Buffered I/O write functions 8–65
bdwrite() 8–65
bawrite() 8–65
bwrite() 8–65
awrite() 8–66

Read write logic 8–67
ncl read 8–68
ncl write 8–70
Raw (physical or direct I/O - physical buffers 8–72
Definition 8–72
Usage 8–72
Buffer assignment 8–72
Physical buffers 8–73
Raw I/O logic 8–74
Physical read 8–74
Physical write 8–76
System buffers 8–78
Definition 8–78
Buffer layout 8–78
Allocation at startup 8–78
Management 8–80
Example 8–82
Device queues 8–82
System buffer logic 8–84
read() logic 8–84
write() logic 8–85
getblk() function 8–86
getbuff() function 8–88
freebuff() function 8–88
bread() function 8–90
bdwrite() macro 8–92
bawrite() macro 8–92
bwrite() function 8–94
Buffer flags 8–96
Asynchronous I/O 8–102
Async I/O buffers 8–102
Initiation 8–104
Completion 8–106
reada(2) and writea(2) system calls 8–108
aread() function 8–112
aiodone() function 8–116
ascomplete() function 8–118
uioreply() function 8–120
List I/0 8–122
listio(2) system call summary 8–122
listio(2) with LC_START 8–124
listio() function 8–126
Pipe file 8–128
Creation 8–128
Allocation 8–130
Pipe data flow 8–130
Writing to pipe 8–132
Pipe write examples 8–134
Write to empty pipe 8–134
Small write to full pipe 8–134

Large write to full pipe 8–135
Write to empty pipe, reader sleeping 8–135
Pipe read examples 8–136
Read from pipe 8–138
Read from empty pipe 8–138
Read from nearly empty pipe 8–139
Read pipe delay processing 8–140
Write pipe delay processing 8–142
Pipe file: Usage by shells 8–144
File management foldouts 8–155
open(2) system call 8–157
read system call logic overview 8–158
write system call logic overview 8–159

I/O Management Subsystem [9] 9–1
Objectives 9-1
Introduction 9.6
Block device driver summary 9-6
Disk drivers – Definition 9-6
Physical drivers 9-6
Logical drivers 9-7
Driver and disks relationships 9-8
/dev directory 9-8
Disk and "special" inode relationships 9-8
Special ldesc files 9-9
Logical type inodes 9-9
Inodes in /dev/dsk 9-9
Logical disk creation – dd devices 9-10
Examples 9-10
Single-slice logical disk 9-10
Multislice logical disk – all dd's 9-12
Two-slice logical disk 9-14
Logical disk creation – Mixed drivers 9-16
Single-slice stripe group 9-16
Logical disk creation – Multislice stripe group 9-18
Logical disk creation – Mirrored logical device 9-20
Logical disk creation – Mirrored stripe group 9-22
Disk drivers – Configuration tables 9-24
slice_prof 9-24
slice tables (lddslice, sddslice, mddslice, pddslice, ssddslice, rddslice) 9-24
sdd_tab 9-25
mdd_tab 9-25
pdd_tab 9-25
ssdd_tab 9-26
hdd_tab 9-26
pdd_prof 9-26
hdd_prof 9-26
pdd_tab 9-29
Block device drivers – Driver overview 9-32
Logical disk drivers 9-32

Physical disk drivers 9-33
Character device drivers 9-35
Interrupt handlers 9-36
Device control 9-40
lddopen/pddopen 9-42
Physical disk I/O example 9-46
Block device specifications 9-69
Disk space 9-71
pddraw – special low-level disk I/O 9-72
pddraw – pddraw() logic flow 9-72
rec_prof() structure 9-78
Basic raw sequence micro instructions #definePASS000/* pass */ 9-79
Raw sequence micro sub-instructions 9-80
Raw sequence micro immediate-instructions 9-81
Error recovery / raw request codes 9-82
Raw sequencer pseudo register definitions 9-83
Disk queue sort 9-84
HIPPI device driver 9-93
HIPPI fundamentals 9-93
lddopen / hddopen 9-96
HIPPI I/O example 9-98
Striped device driver 9-118
lddopen / sddopen 9-118
Striped I/O example 9-122
Mirrored device driver 9-164
lddopen / mddopen 9-164
Mirrored I/O example 9-168
SSD driver 9-190
lddopen() 9-190
lddopen / ssddopen 9-192
SSD related structures 9-194
RAM device driver 9-196
lddopen / rddopen 9-196
Channel drivers 9-201
IOS packet driver – packet tables 9-201
IOS packet driver – packet retransmission 9-204
Definition 9-204
Mainframe retransmission 9-204
IOS retransmission 9-204
Packet interrupt handler table 9-205
VHISP driver – VHISP driver's place in disk management 9-207
VHISP driver – data structures 9-208
VHISP driver – high level 9-210
ssddstart 9-210
VHISP driver – low level 9-212
vhispio 9-212
ssdchk 9-212
Secondary data segments – SDS data structures 9-214
SDS logic 9-214
SDS usage and space management 9-214

Logical device caching 9-221
ldcache buffers 9-222
Definition and usage 9-222
Buffer layout 9-222
ldch_io() 9-224
ldch_iofini() 9-226
ldcache I/O structures 9-228
ldcache examples 9-230
Recent ldcache options 9-236
ldcache internal processing hierarchy 9-237
Character devices 9-239
Introduction – character device driver summary 9-240
Generic character device 9-248
Terminal device driver – tty/pty tables 9-250
tp_tty and pt_tty 9-250
Terminal device driver – character buffers 9-252
Terminal device driver – character queues – clists (tty) 9-254
tty queues 9-254
Data movement (tty devices using T packets) 9-254
Relationship of a user's file descriptor to the terminal buffers 9-254
pty queues 9-256
Data movement (pty devices using N packets) 9-256
Terminal device driver – clist processing 9-258
Use of the cblock clist 9-258
Terminal device driver – pseudo terminals 9-260
Terminal device driver – session controlling tty 9-261
Establishing an interactive session 9-261
Terminal special character processing 9-261
/dev/tty access to the terminal 9-261
Exit processing 9-261
Terminal device driver – pty – user read of 9-263
Terminal device driver – pty – user write to 9-265
Sockets 9-268
Socket read 9-268
Socket interrupt 9-269
Socket write 9-270
Socket structures 9-271
Mbuf management 9-272
Mbuf headers and mbuf data area 9-272
Mbuf queues and queue pointers 9-276
Mbuf initialization 9-280
Mbuf allocation 9-280
Mbuf structure initialization 9-280
Mbuf allocation logic (_m_gets) 9-283
Compute number of mbuf data segments needed 9-283
Search mbuf queues for preallocated space 9-283
Search mbuf free chains 9-283
If mbuf space found 9-283
If sufficient contiguous mbuf space not found 9-284

Mbuf deallocation logic (_m_free) 9-286
Compute number of mbuf data segments being released 9-286
If queue size match, put on mbuf queue 9-286
If not queue size match, put on mbuf free chains 9-286
Wakeup any thread waiting for mbuf space 9-286
BMX tape driver – tape components 9-288
BMX tape driver – tape daemon tables 9-290
BMX tape driver – tape special files 9-291
BMX tape driver – tape files 9-292
BMX tape driver – sample tape session 9-293
BMX tape driver – avrproc 9-298
BMX tape driver – user overview 9-300
Definitions 9-300
Setup 9-301
BMX tape driver – system cache buffer allocation 9-302
BMX tape driver – programming considerations 9-303
BMX tape driver – transparent buffered 9-304
BMX tape driver – transparent unbuffered 9-306
BMX tape driver – tape list I/O (single list) 9-308
BMX tape driver – tape list I/O (multiple list) 9-310
BMX tape driver – packet summary 9-312
Command list entries 9-312
BMX tape driver – bmxread/bmxwrite 9-314

Dump Analysis [10] 10–1
Objectives 10–1
Dump analysis component description 10–2
System dump core file creation 10–4
Configuration for a system dump 10–6
Dump device configuration 10–6
param file sample sections 10–7
Mainframe dump controlling parameters 10–8
Initializing the dump device 10–10
Dump device size 10–12
UNICOS panic 10–14
Panic buffer 10–14
UNICOS panic logic 10–16
Panic calls 10–18
Panic routine 10–20
Assert 10–22
Dumping the UNICOS system 10–24
dumpsys 10–24
(x)dumpsys 10–24
(x)dumpsys 10–24
mfboot 10–24
mfsysdmp 10–26
mfsysdmp 10–26
System dump core file creation 10–28
Model D IOS differences 10–30
Dump device configuration 10–30

Dump device initialization 10–30
Dumping the UNICOS system 10–30
System dump core file creation 10–30
Using crash(8) 10–32
crash(8) man page 10–33
crash(8) help output 10–45
crash command / source cross-reference 10–48
Low memory tables and pointers 10–53
Exchange packages 10–53
Fixed location communication area 10–53
lowmemtbl 10–54
Panic buffer and kernel trace table 10–56
Selected tables and pointers 10–56
ucomm structure debugging aids 10–62
us_ascii location and dumping 10–62
crash command 10–64
Table/structure relationships 10–64
Model E IOS 10–66
Model B/C/D IOS 10–68
proc 10–70
od 10–74
kfp 10–77
trace 10–78
ut 10–80
CPU / process context summary 10–82
Getting started on a system dump 10–84
Kernel trace messages 10–92
CAL trace entries 10–92
C language trace entries 10–94
Compiling selected kernel modules for debugging purposes 10–135

UNICOS 9.0 Differences [A] A–1
UNICOS kernel changes to support CRAY T90 hardware A-2
System control hardware differences A-2
System name A-4
System numbers, mainframe type, and subtypes A-4
Expanded exchange package A-5
Enlarged address registers A-6
Revised sizes and maximums A-6
Native "Triton" mode versus compatibility C90 mode A-7
Bit matrix multiply A-7
Kernel usage of the logical address translation (LAT) registers A-8
User usage of the logical address translation (LAT) registers A-9
Displaying shared memory with /etc/crash A-10
Configuration of shared memory A-11
Configuration of IPC semaphores and IPC messages A-11
User usage of the scalar cache A-13
Cache incoherency example A-13
Kernel usage of the scalar cache A-14
New pending I/O interrupt detection mechanism A-14

Multiple exit address registers A-16
IEEE CPUs A-16
Holding of clusters A-16
Register parity error and memory error handling A-17
New VHISP done flag A-17
New LSOP parity error flag A-17
Programmable interrupt (PINT) channel A-17
Pthreads model A-18
Change in terminology A-18
System call changes for Pthreads A-18
Change in `getpid`(2) A-18
Old `getpid`(2) compatibility A-18
Change in `exit`(2) A-18
Old `exit`(2) compatibility A-18
Change in `getppid`(2) A-20
Old `getppid`(2) compatibility A-20
Change in `waitpid`(2) A-20
Old `wait`(2) compatibility A-20
Change in `kill`(2) A-22
Old `kill`(2) compatibility A-22
Change in `killm`(2) A-22
Old `killm`(2) compatibility A-22
Change in `alarm`(2) A-22
Old `alarm`(2) compatibility A-22
Asynchronous signals in general A-23
Library system calls: `lib/libc/sys` A-25
User-level fair share A-27
The `clock()` function A-30
Excess user errors controlled A-32
Ill-formed I/O split into physical and buffered chunks A-34
Example of a split I/O request A-36
Performance improvement examples A-38
Performance degradation example A-40
Dynamic allocation of the `NPBUF` parameter added A-41
Implications of `NPBUF` A-41
`csim` utility no longer supported A-42
New character device drivers A-42
CRAY T3D phase II I/O (backdoor) A-44
CRAY T3D system activity monitoring A-45
Dynamic allocation of file descriptors A-46
Initial file descriptors A-46
Additional file descriptors A-46
Logical `fd` limit A-47
Other uses of `kmem` A-47
Configuration of `kmem` A-47
Maximum number of current primary partitions A-50
Example 1: default to 4 primaries; all current A-51
Example 2: specify 5 primaries; all current A-51
Example 3: specify all 9 slices as primaries; all current A-52
Effect on pre-9.0 file systems A-53

New crash directives for UNICOS 9.0 (and 8.3) A-54
SFS processes for 9.0 .... A-63
Move of XPs to xpa A-63

CRAY T90 Hardware Configuration [B] B–1
T932 support system and mainframe components B-2
The operator workstation (OWS-T) B-2
The maintenance workstation (MWS-E) B-2
The support system VME chassis B-4
The maintenance channel B-4
The boundary scan channel B-4
The support channel B-5
The CRAY T90 mainframe configuration B-6
Kernel support of logical partitioning B-9
Logical machine “machine environment” file B-9
Dynamic reconfiguration B-9
Support of UNICOS under UNICOS B-10
Startup differences B-11
Boot with RAM root B-11
Role of the OWS B-11
Preserving previous system state B-12
CSL extension B-12
Memory degrading and partitioning B-13
Memory degrading example B-15
Memory degrading special case B-16
Memory grouping B-17
Partitioning by I/O grouping and memory groups B-19
Partitioning by memory degrading B-20
T916 support system and mainframe components B-22
T916 component summary B-23
T916 memory layout B-24
T916 memory configuration B-25
T94 support system and mainframe components B-26
T94 component summary B-27
T94 memory layout B-28
T94 memory configuration B-29

Preface

This training document provides reference material for UNICOS Kernel Internals and UNICOS Dump Analysis classes.

Software Education Catalog

The Software Education Catalog is now available online on the World Wide Web (WWW). To access the online catalog, use the following URL:

```
http://www.cray.com/PUBLIC/CUSTSERV/SW-ED/
```

To obtain a hardcopy version of this catalog (TR–CUSTCAT), contact your Training Registrar, call the Distribution Center at 1–612–683–5907, or send a facsimile of your request to fax number 1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Conventions

The following typographic conventions are used throughout this training document:

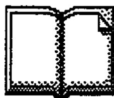
Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:

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

Convention	Meaning
routine()	Routine names followed by an empty set of parentheses designate a library or kernel routine; for example, `ddcntl()`. Kernel routines do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown is nonbold, fixed-space font.
`abbreviation`	Underlining indicates the shortest possible abbreviation for a command.
[]	Brackets enclose optional portions of a command line.
...	Ellipses indicate that a preceding command-line element can be repeated.
KEY	This convention indicates a key on the keyboard.
<KEY>	On man pages, this convention indicates a key on the keyboard.

The following icon conventions are used throughout this training document:

Note icon. A *note* highlights items of particular interest and essential operating or maintenance procedures, conditions, and statements.



Book icon. This highlights sources of other information that may be beneficial to students.

Caution icon. A *caution* highlights actions that could cause extreme inconvenience to users, destroy data, or produce unpredictable results.

Warning icon. A *warning* highlights actions that could harm people or could damage equipment or system software.

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

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems, including the following: CRAY C90 series (CRAY C916, CRAY C92A, CRAY C94, CRAY C94A, and CRAY C98 systems) CRAY C90D series (CRAY C92AD, CRAY C94D, and CRAY C98D systems) CRAY EL series (CRAY Y-MP EL, CRAY EL92, CRAY EL94, and CRAY EL98 systems) CRAY J90 series (CRAY J916 and CRAY J932 systems) CRAY T90 series (CRAY T94, CRAY T916, and CRAY T932 systems) CRAY Y-MP E series (CRAY Y-MP 2E, CRAY Y-MP 4E, CRAY Y-MP 8E, and CRAY Y-MP 8I systems) CRAY Y-MP M90 series (CRAY Y-MP M92, CRAY Y-MP M94, and CRAY Y-MP M98 systems)
Cray MPP systems	All configurations of Cray massively parallel processing (MPP) systems, including the CRAY T3D series (CRAY T3D MC, CRAY T3D MCA, and CRAY T3D SC systems)
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release
SPARC systems	All SPARC platforms that run the Solaris operating system version 2.3 or later

The default shell in the UNICOS 9.0 release, referred to in Cray Research documentation as the standard shell, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Company Standard XPG4

In this training document, *Cray Research*, *Cray*, and *CRI* refer to Cray Research, Inc. and/or its products.

Man page sections

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

Section heading	Description
NAME	Specifies the name of the entry and briefly states its function.
SYNOPSIS	Presents the syntax of the entry.
IMPLEMENTATION	Identifies the Cray Research systems to which the entry applies.
STANDARDS	Provides information about the portability of a utility or routine.
DESCRIPTION	Discusses the entry in detail.
NOTES	Presents items of particular importance.
CAUTIONS	Describes actions that can destroy data or produce undesired results.
WARNINGS	Describes actions that can harm people, equipment, or system software.
ENVIRONMENT VARIABLES	Describes predefined shell variables that determine some characteristics of the shell or that affect the behavior of some programs, commands, or utilities.
RETURN VALUES	Describes possible return values that indicate a library or system call executed successfully, or identifies the error condition under which it failed.
EXIT STATUS	Describes possible exit status values that indicate whether the command or utility executed successfully.
MESSAGES	Describes informational, diagnostic, and error messages that may appear. Self-explanatory messages are not listed.
FORTRAN EXTENSIONS	Describes how to call a system call from Fortran. Applies only to system calls.
BUGS	Indicates known bugs and deficiencies.
EXAMPLES	Shows examples of usage.

Section heading	Description
FILES	Lists files that are either part of the entry or are related to it.
SEE ALSO	Lists entries and publications that contain related information.

Online information

1.1

- CrayDoc online documentation reader, which lets you see the text and graphics of a manual online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Docview text-viewer system, which lets you see the text of a manual online. The Docview system is available on the Cray Research mainframe. To start the Docview system, use the `docview`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a manual. To get a definition, use the `define`(1) command.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

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

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!TR-ITR`

- Send us electronic mail from any system connected to Internet, using the following Internet addresses:

  `TR-ITR@timbuk.cray.com` (comments on this manual)

- Call our Software Education Services department in Eagan, Minnesota, through the Technical Support Center, using either of the following numbers:

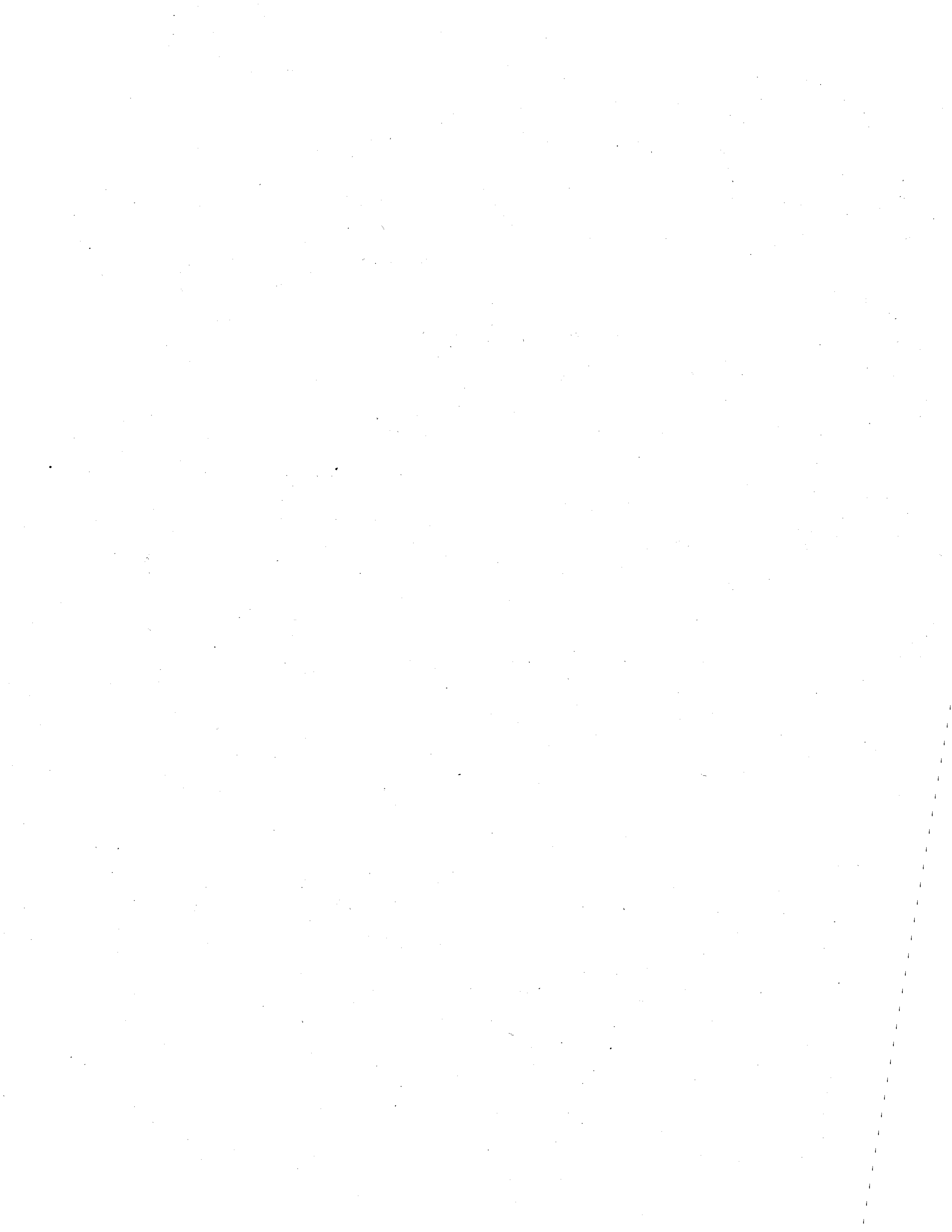
  1–800–950–2729 (toll free from the United States and Canada)

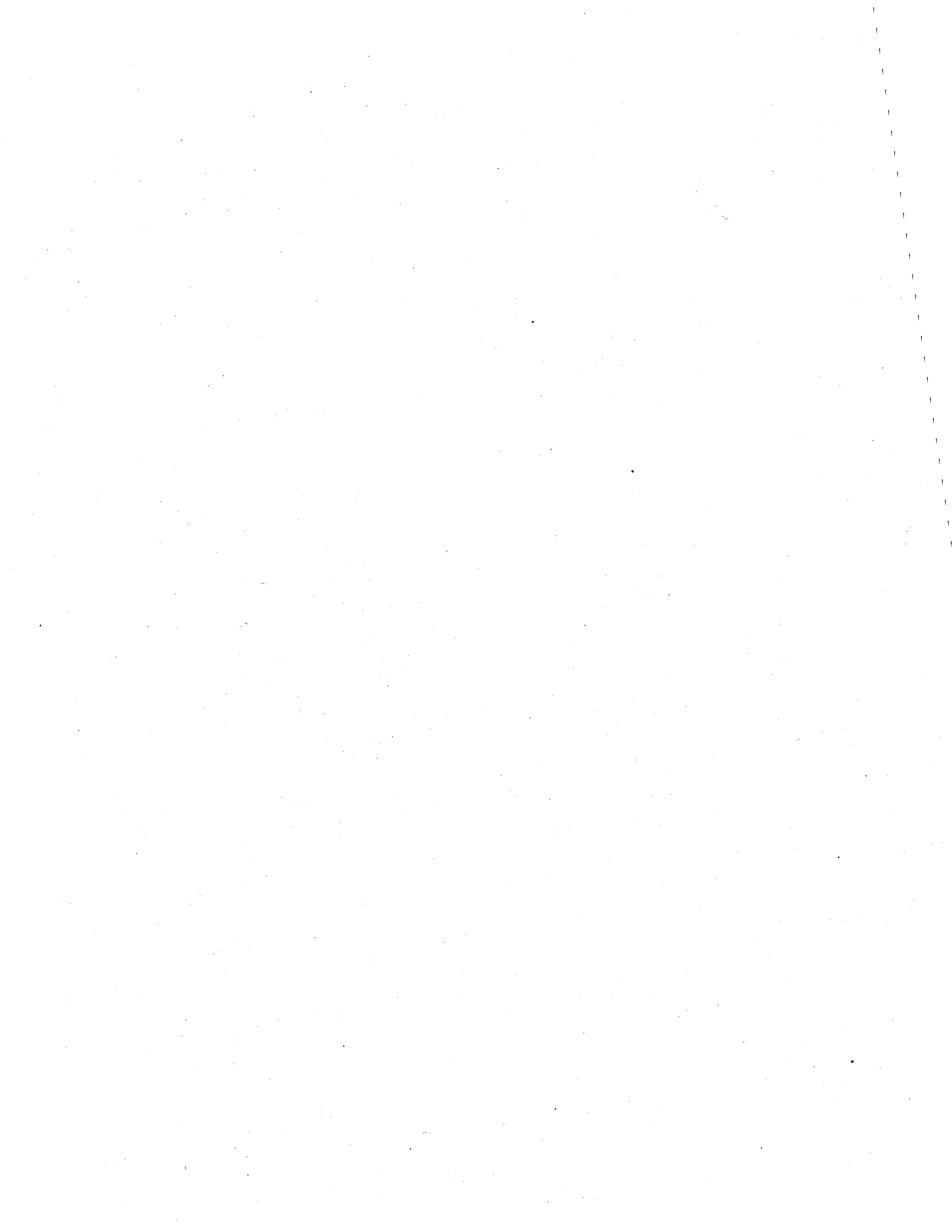
  1–612–683–5600

- Send a facsimile of your comments to the attention of "Software Education Services" in Eagan, Minnesota, at fax number 1–612–683–5599.

- Use the postage-paid Reader's Comment Form at the back of this training document.

We value your comments and will respond to them promptly.





Contents

Software Overview [1] 1–1

Objectives 1–1
Overview 1–2
Recommended reading 1–3
Recommended outside reading 1–4
UNIX bibliography 1–4
System functional overview 1–6
  Session from a real terminal 1–6
  Session from a pseudo terminal 1–8
  Interactive session from a Cray station 1–10
  Batch access from a Cray station 1–12
Kernel organization overview 1–14
  System memory organization 1–14
  Source code organization 1–16
    System-wide files 1–16
    UNIX time-sharing system `uts/` (kernel source code) 1–17
  Object code organization 1–20
  Kernel logical organization 1–22
  Kernel structures for process control 1–26

Software Overview [1]

Objectives

After completing this section you should be able to:

- **Reference a list of sources about UNIX and UNICOS**
- **Describe the kernel's functional role in the entire UNICOS scheme**

  Summarize the components in an interactive sessions

  Summarize the components in processing a batch job
- **Describe these aspects of the kernel:**

  Placement in memory

  Source code tree

  Object modules

  Logical components block diagram

Overview

The "System Initialization" chapter provides a general overview of the kernel.

Lists of UNICOS manuals related to studying the kernel and general reference material is provided.

Diagrams present an overview of UNICOS and show the relations of the major software systems to the UNICOS kernel.

The UNICOS source tree is resented with detail on the major directories and files used to build the UNICOS kernel.

The organization of the primary components of the kernel with their relationships to the rest of UNICOS is presented in diagram form.

Sample memory maps are provided to show the key elements in the UNICOS system, their position in memory, and relative sizes in memory.

Recommended reading

The following information supplements material in this section:

Manual title	Publication number
UNICOS	
Index for Cray Y-MP, EA, X-MP, and Cray-1 Computer Systems	SR–2049 8.0
UNICOS Commands Reference Manual	SR–2011 8.0
UNICOS System Calls Reference Manual	SG–2012 8.0
UNICOS System Calls	TR–USC 8.0
TCP/IP and OSI Network User Guide	SG–2009 8.0
Administration / Internals	
UNICOS System Administration	SG–2113 8.0
UNICOS Administrator Commands Reference Manual	SR–2022 8.0
UNICOS Source Manager (USM) User's Guide	SG–2097 8.0
UNICOS Kernel Structures	TR–UKS 8.0
Loaders / Libraries / Languages	
CAL Assembler Version 2 Reference Manual	SR–2003 2.0
Segment Loader (SEGLDR) and ld Reference Manual	SR–0066 8.0
Cray Standard C Reference Manual	SR–2074 4.0
Volume 6: UNICOS Internal Library Reference Manual	SM–2083 8.0
Other	
UNICOS File Formats and Special Files Reference Manual	SR–2014 8.0
UNICOS Support Tools Guide	SG–2016 8.0
UNICOS Tape Subsystem User's Guide	SG–2051 8.0
UNICOS NFS Internal Reference Manual	SM–2065 8.0
SUPERLINK UNICOS Reference Manual	SI–0185 8.0
CRAY Hardware	
CRAY X-MP/1 System Programmer Hardware Reference Manual	CSM–0111000
CRAY X-MP/2 System Programmer Hardware Reference Manual	CSM–0110000
CRAY X-MP/4 System Programmer Hardware Reference Manual	CSM–0112000
CRAY Y-MP System Programmer Reference Manual	CSM–0400–0A0
CRAY Y-MP C90 System Programmer Reference Manual	CSM–0500–000

Recommended outside reading

The following information supplements material in this section:

User-level UNIX:

Sobell, Mark G. *A Practical Guide to UNIX System V* (see bibliography)

Operating system design:

Bach, Maurice J. *The Design of the UNIX Operating System* (see bibliography)

Leffler, Samuel J. [et al.] *The Design & Implementation of 4.3 BSD UNIX* (see bibliography)

Comer, Douglas. *Operating System Design: The XINU Approach* (see bibliography)

Tanenbaum, Andrew S. *Operating Systems Design and Implementation.* Prentice-Hall, 1987.

IEEE Computer Society Portable Operating System Interface for Computer Environments (POSIX), Institute of Electrical and Electronic Engineers, Inc. 1988.

Fair-share scheduler:

Kay, J. and Lauder, P. A Fair Share Scheduler Communications of the ACM January 1988

UNIX Bibliography

Anderson, Gail and Paul Anderson. ***The UNIX C Shell Field Guide.*** Prentice-Hall, Inc., 1986.

Arthur, Lowell Jay. ***UNIX Shell Programming.*** Wiley, 1986.

AT&T Bell Laboratories Technical Journal. *UNIX System.* October, 1984.

Bach, Maurice J. ***The Design of the UNIX Operating System.*** Prentice-Hall, Inc., 1986.

Bell System Technical Journal. ***UNIX Time-Sharing System.*** July/August, 1978. Vol. 57, No. 2.

Bourne, S. R. ***The UNIX System.*** Addison-Wesley, 1983.

Comer, D. ***Operating System Design: The XINU Approach.*** Prentice-Hall.

Derman, Bonnie ed. ***Applied C.*** Strawberry Software, Inc., Van Norstrand Reinhold Co., N.Y., 1986.

Feuer, A. R. ***The C Puzzle Book.*** Prentice-Hall, 1982.

Foxley, E. ***UNIX For Super-users.*** Addison-Wesley, 1985.

Groff and Weinberg. ***Understanding UNIX: A Conceptual Guide.*** Que Corp., 1983.

Harbison and Steele. ***C: A Reference Manual.*** Prentice-Hall, 1984.

Kernighan, Brian W. and Rob Pike. ***The UNIX Programming Environment.*** Prentice-Hall, Inc.,1984.

Kernighan, Brian W. and Dennis M. Ritchie. ***The C Programming Language***. Prentice-Hall, 1978.

Kochan, Stephen G. ***Programming in C.*** Hayden Book Company, 1983.

Leffler, Samuel J. [et al.] ***The Design & Implementation of 4.3 BSD UNIX*** Addison-Wesley, 1989.

Mcgilton, H. and Rachel Morgan. ***Introducing The UNIX System.*** McGraw-Hill, 1983.

Plum, T. ***C Programming Guidelines***. Plum-Hall.

Plum, T. ***Learning To Program In C***. Plum-Hall.

Prata, Stephen. ***Advanced UNIX – A Programmer's Guide.*** Howard W. Sains & Co., Inc.,1985.

Rochkind , Marc J. ***Advanced Unix Programming.*** Prentice-Hall, Inc., 1985.

Schildt, Herbert. ***C: The Complete Reference***. Osborne McGraw-Hill, 1987.

Sobell, Mark G. ***A Practical Guide to UNIX System V.*** The Benjamin/Cummings Publishing Company. 1985.

UNIX Time-Sharing System: ***UNIX Programmer's Manual Seventh Edition***. Volume 2. Holt, rt and Winston.

Waite, Mitchell and Prata, Stephen and Martin, Donald. ***C Primer Plus***. The Waite Group, Howard W. Sams & Co., 1987.

Wood, Patrick H. and Stephen G. Kochan. ***UNIX Shell Programming.*** Hayden Book Company, 1985.

Wood, Patrick H. and Stephen G. Kochan. ***UNIX System Administration.*** Hayden Book Company, 1985.

Wood, Patrick H. and Stephen G. Kochan. ***UNIX System Security.*** Hayden Book Company, 1985.

System functional overview

Session from a real terminal

Real terminals are few in number in UNICOS. They are windows on the Operator Work Station (OWS) on systems with an IOS Model E or on systems with an IOS Model B, C, or D (without an OWS), the operator's consoles attached directly to the Master I/O Processor (MIOP).

UNICOS monitors its real terminals in the classical UNIX way: by keeping a getty process active for each terminal. The `init` daemon forks a `getty` process for each terminal at system initialization time, as directed by `/etc/inittab` file.

The `getty` process opens a real terminal character device, thus establishing the pathway into the kernel. It displays a login prompt and then sleeps on a read from that terminal.

The session begins when a user enters his uid at the terminal. The IOS terminal driver reads characters from the terminal and packages them for the mainframe kernel.

All data and control information between the IOS driver and the kernel's driver are exchanged via packets transferred on the low-speed channel.

The kernel terminal driver delivers the input to the `getty` process, completing its read and awakening the `getty` process.

The `getty` command becomes the login command (by an `exece`() system call) and validates the user. Then the `login` command becomes the user's `login` shell.

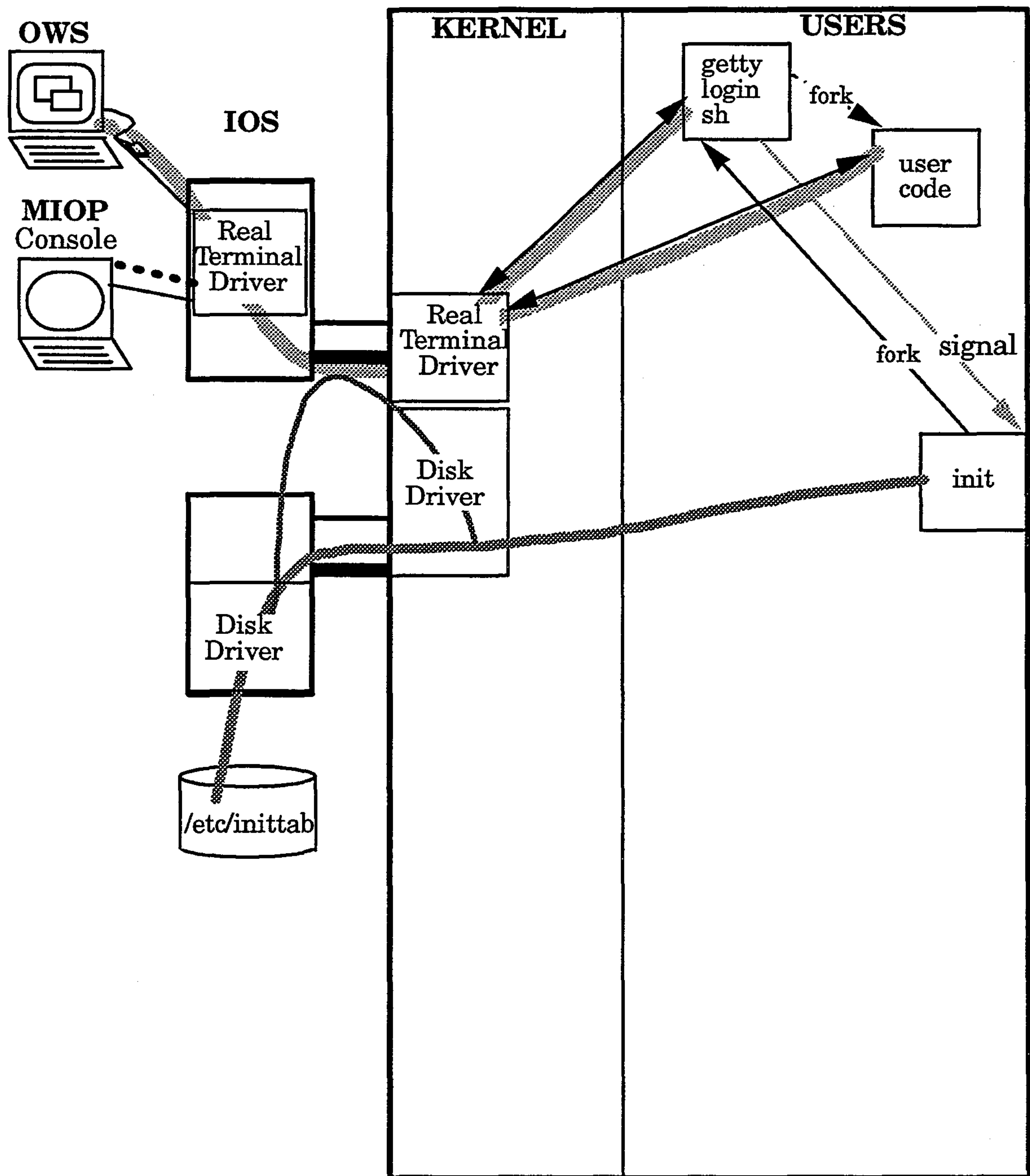
User commands received by the shell result in `fork`() and `exece`() calls to execute the user's binary codes.

The `login` shell exits the system when the user logs off. The `init` process, the parent of the `login` shell, receives a death-of-child signal and is responsible for recreating the `getty` process by issuing another fork request.

The function of the kernel disk driver is shown here as well. It is entered on `read`() system calls from the `init` daemon in this picture, as `init` reads its directives file.

The kernel disk diver communicates control information to the IOS disk driver via the low-speed channel. Disk data is transferred using the high-speed channel.

Session from a Real Terminal




```
$ grep getty /etc/inittab
```

Session from a pseudo terminal

Pseudo terminals are the usual interface between a user on a remote system and a process running in UNICOS. In this case there is no getty process, just a daemon (inetd) waiting for requests for service to arrive from a remote system.

The inetd daemon has several TCP/IP control socket files opened to the kernel. Each control socket has a different port number, corresponding to a different service (see the file /etc/services).

When a user executes telnet command on the remote system, that system sends a message through the IOS network driver to the kernel network driver, which in turn hands it off to TCP/IP routines in the kernel. The message asks for telnet service, so TCP constructs a data socket file to the inetd daemon and returns control to inetd.

The inetd daemon receives the data socket file descriptor from its telnet control socket so it forks a telnet daemon (telnetd), closes its access to the data socket, and goes back to its job of listening to its control sockets. The child of the inetd daemon becomes the telnetd command, and can communicate data to/from the remote user via the socket.

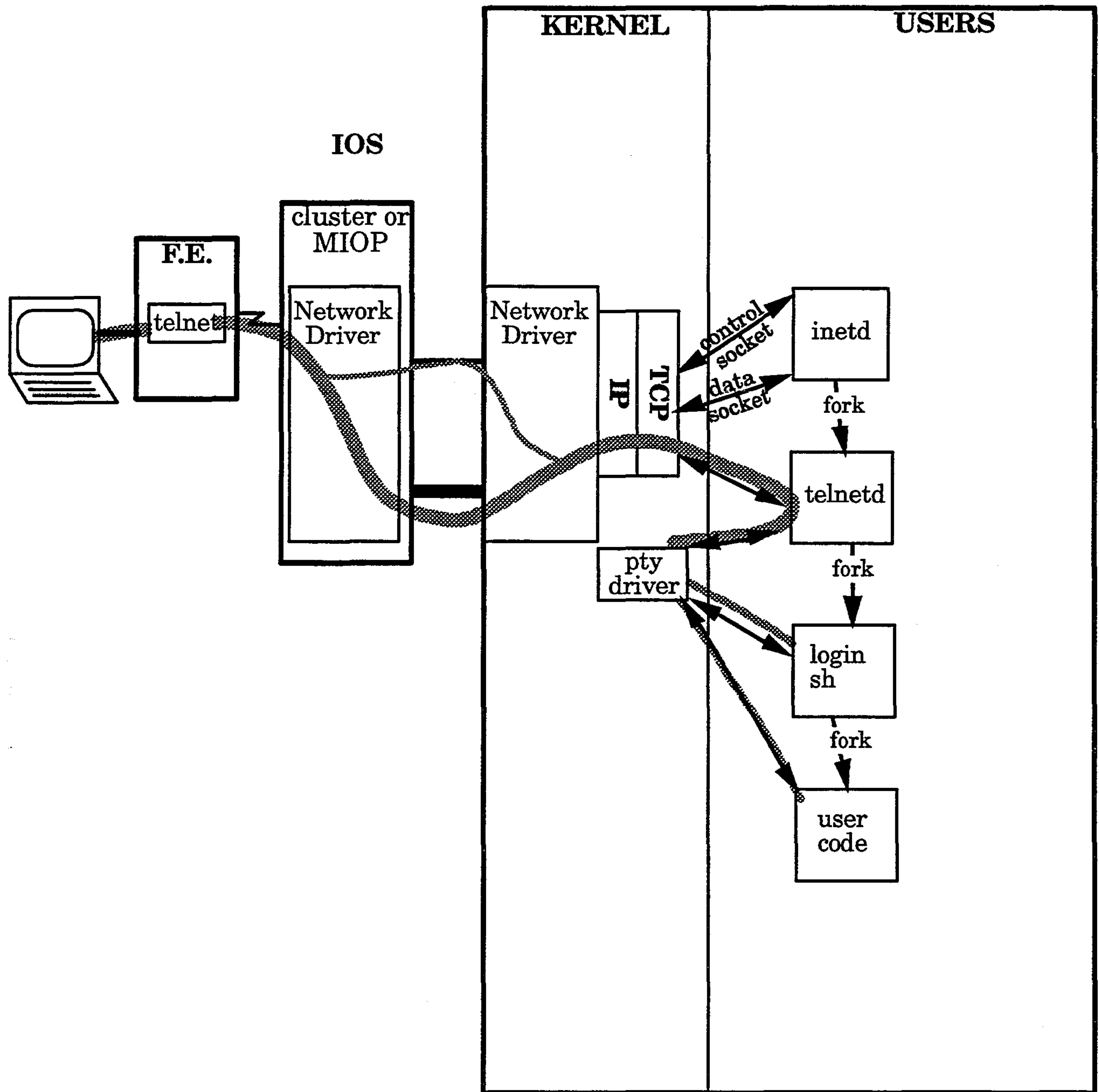
This data must be filtered through standard kernel terminal processing logic, and communicated with a login shell, so the telnet daemon opens a pseudo terminal file pathway into the kernel. This is a character device driver in UNICOS. The telnetd daemon forks a child which opens the master and then the slave side of the pseudo terminal and then becomes the login/shell.

(In prior releases of UNICOS the device number of this pseudo terminal (pty) was communicated to the init daemon via a named pipe along with the request to fork a login / shell. Init forked the child process. The init child process opened the user side of the pseudo terminal (by the number received from telnetd). Then the init child became the login command. The rlogind, uscpd and rshd still use this method.)

The login process communicates with the user via the pseudo terminal in exactly the same manner as it would using a real terminal. When the login process has validated the user, it becomes the login shell. The shell forks user commands as requested.

When the shell exits, a death-of-child signal is returned to the inetd daemon process. The telnetd process exits because it detects the closed slave side of the pty.

Session from a Pseudo Terminal




```
$ grep telnet /etc/services
$ls -l /dev/pty | pg
```

Interactive session from a Cray station

A remote user may log into UNICOS through most Cray front-end station software. UNICOS must be running the UNICOS station call processor daemon (`uscpd`).

The station communicates with the `uscpd` daemon using a Cray Research proprietary station protocol, and thus TCP/IP is not involved. The `uscpd` daemon opens a character device pathway between itself and each of the stations.

On receipt of a station's request to establish an interactive session, the `uscpd` daemon peforms the following:

- Opens the master side of a pseudo terminal.
- Then writes a request to the init daemon (on the named pipe) to create a login shell for the slave side of this particular pseudo terminal. *(This is the same method used by* `the rlogind` *command, and used by* `telnetd` *command prior to release 7.0)*

The `init` process calls `fork` to create a child process.

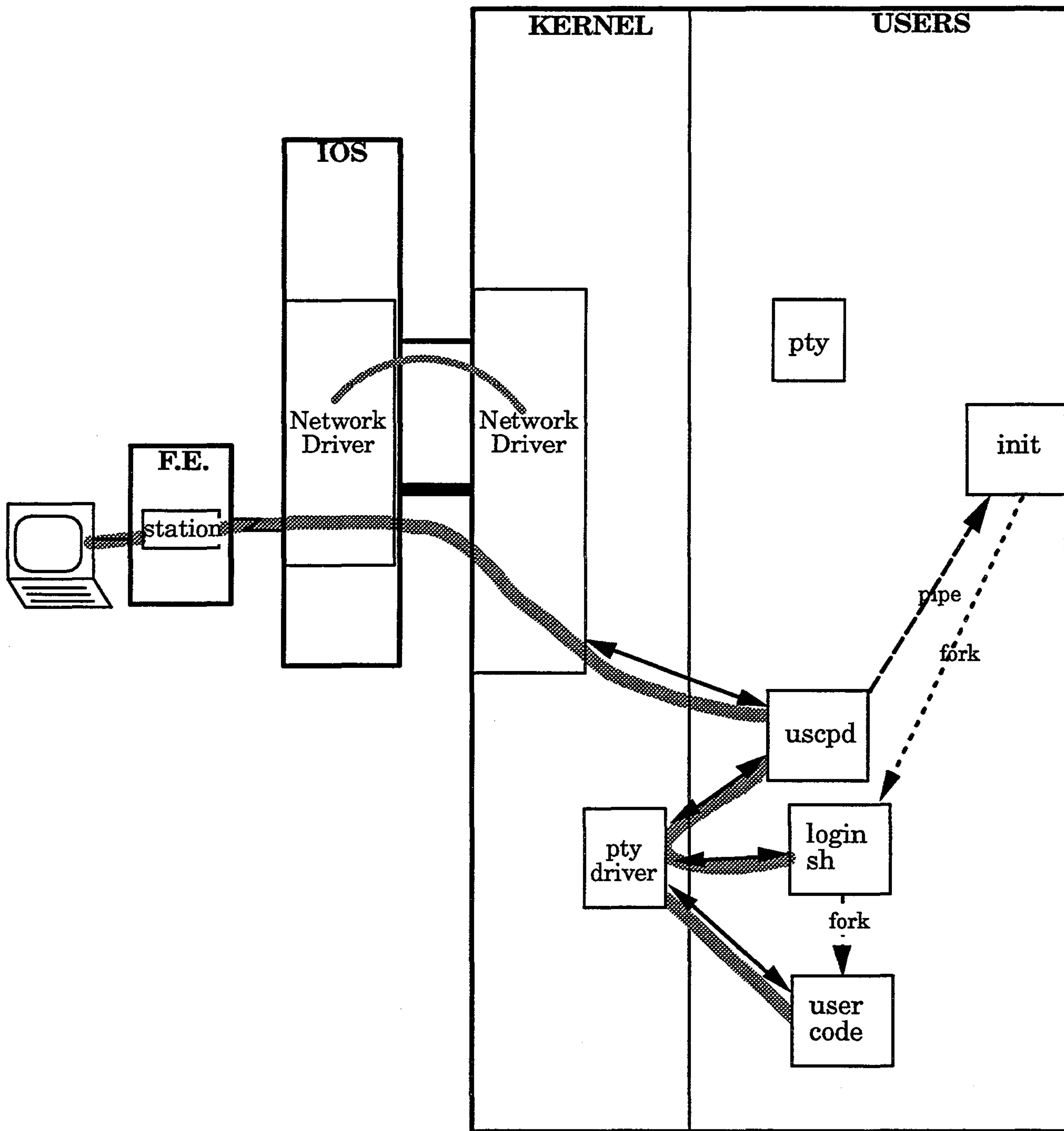
The `init` child process performs the following:

- Opens the slave side of the pseudo terminal.
- Becomes the `login` command.

All terminal input and output is funneled through the `uscpd` daemon.

- The `uscpd` daemon unpackages input from the station and writes it to the pseudo terminal as raw character strings.
- The `uscpd` daemon packages the user output it receives from the pseudo terminal and communicates it to the station using the proprietary station protocol.

Interactive Session from a Cray Station



Batch access from a Cray station

A remote user using a Cray front-end station may submit a front-end resident script for batch execution, and receive its output on the front-end. Batch execution requires the network queuing subsystem (NQS) to be running under UNICOS.

The script is prepared on the front-end, including any desired directives for NQS such as uid and resource requirements. A command to the Cray station causes the script to be sent to the `uscpd` daemon using the proprietary Station Call protocol. A header is also sent, describing where the output of the job is to be returned.

The `uscpd` daemon saves the job header and writes the job script to one of its own directories using standard user-level I/O. It then assigns the task of submitting the script to NQS to its companion daemon `uscpcmd`.

When (and if) NQS is running, the uscpcmd daemon becomes the user (via setgid(2)/`setuid`(2) requests) and forks a `qsub` command to submit the script file to NQS.

NQS queues the script according to its own queuing scheme.

When NQS schedules the script for execution, it forks a shepherd process whose job it is to report back to NQS when the "job" is done.

The shepherd process forks a child process. The child process creates a "job" for itself by making a system call request, makes the script its standard input file, redirects standard out and standard error to a disk file, and then become the `login` shell.

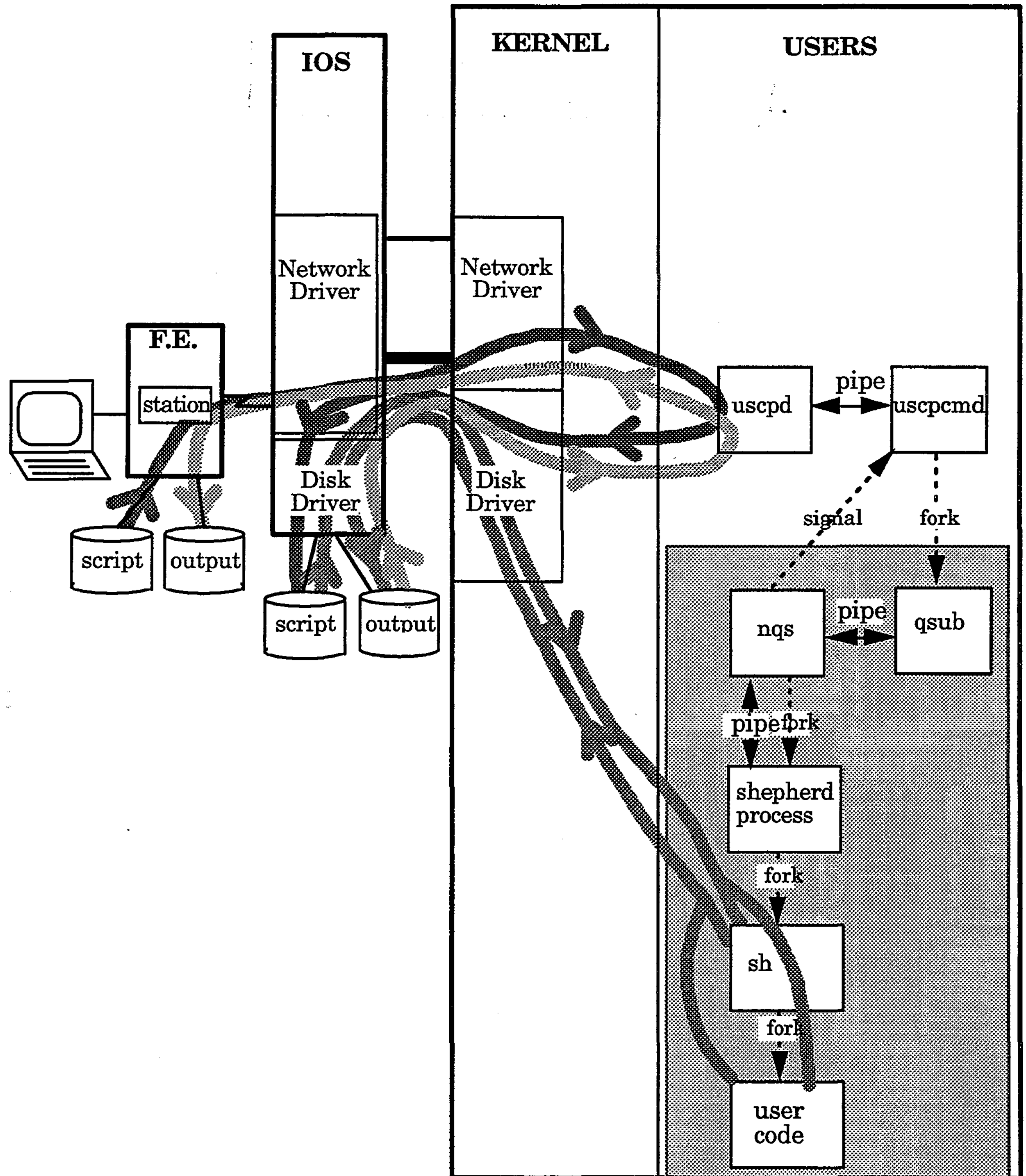
The shell executes the script, forking to execute user codes as directed.

On death of the `login` shell, a signal is returned to the shepherd process, which notifies NQS of the job's completion. The script is deleted.

A signal is sent from NQS to `uscpcmdm` daemon, which notifies the `uscpd` daemon that the job output should be sent back to its originating station.

The `uscpd` daemon sends the output file back to the station along with the header telling the station where to store it. The `uscpd` daemon then deletes the job's output file.

Batch Access from a Cray Station



Kernel organization overview

System memory organization

Not Updated 8.0

The figure on the right shows the relative placement and memory usage of the major parts of a large configuration system (sn1920; CRAY-YMP4/64).

Memory sizes are shown in octal blocks (a block is 512 words). *(Notice that the boot kernel itself may be a relatively small part of total system memory.)*

Large portions of kernel memory are as follows: (with significant variability from one configuration to another)

- Kernel tables: 4 CPUs, 600 procs, 300 sessions, 2200 inodes, 2200 files, 256 blocked logical devices
- Memory disk (This memory is allocated only if a memory-resident RAM disk is configured.)
- Run time kernel code (This does not include some startup code whose space is reused; full blown kernel includes NFS except for the `ipi3` driver.)
- System `bufs`/cache: 3000 blocks, 2048 hash lists
- `ldcache` headers (6750 units of ldcache are possible on systems with an SSD, BMR and /or central memory.)
- Central memory ldcache (A bit map and memory allocation are made only if the `LDCHCORE` bit is set.)
- TCP/IP buffers: 4000 1024-byte message buffers
- Relatively small allocations of kernel memory are as follows:

Disk configuration tables:	Typically, one entry per physical disk device
Dump memory descriptors:	2 words per possible process; used by `sysdump`
Asynio's:	400 asynchronous I/O operation headers
Exec arguments:	Storage for 50000 bytes during `exece`(2) calls
NFS rnodes:	256 NFS file-system specific inodes
Restart tables:	4 entries; used for process restart
Sidedoor buffer:	50-block buffer for SDS to SSD file-system transfers and where no backdoor exits

System Memory Organization

Address	Memory region	blocks	(octal)
0	kernel tables	2565	19.2%
/unicos	run-time kernel code	1214	8.9%
	disk config. tables	422	3.8%
	[ram disk] OPTIONAL	0	
	dump mem. descriptors	2	
	cache bufs	207	1.9%
	cache buffers	5670	41.2%
	cache hash table	30	
	cache hash bufs	44	
	asynio's	22	
	exec arg. pool	20	
	ldcache headers ← OPTIONAL	646	5.8%
	[central memory ldcache] ← OPTIONAL	0	
	nfs rnodes	107	.1%
	~~superlink buf's~~	4	
	~~superlink f.s. buffers~~	22	
	~~superlink ipc buf's~~	11	
	restart tables	31	
	TCP mbuf's	175	1.7%
	TCP buffers	1750	13.7%
3.8 MWords	sidedoor buffer	62	
		16156_8	

Handwritten note at the top of the memory diagram: lowmem.c

Disk config. tables through cache buffers: allocated during system initialization

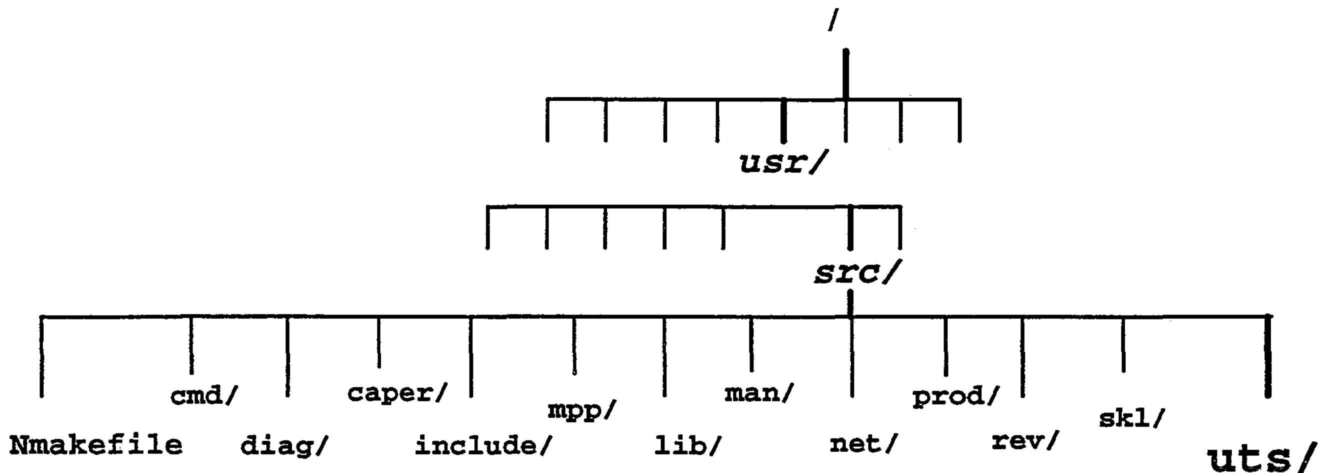
Not Updated 8.0

Source code organization

System-wide files

Nmakefile	Make file that can execute the make file in any /usr/src subdirectory
cmd/	Source for standard commands such as ls(1), ps(1), pwd(1), cat(1) and so on
diag/	Source for on-line diagnostics
caper/	Generation tools
include/	System-wide include (.h) files
mpp/	MPP support code
lib/	Source for UNICOS libraries such as libio(3), libsci(3), libc(3), and so on
man/	Man pages source
net/	Source for user level network software including X11 (x windows), NFS, NQS, RCP, and USCP
prod/	UNICOS commands such as update(1), usm(1), segldr(1), scc(1), pcc(1), and so on
rev/	Revision modifications
skl/	Scripts for system administration
uts/	**Kernel source code**

Source Code Organization

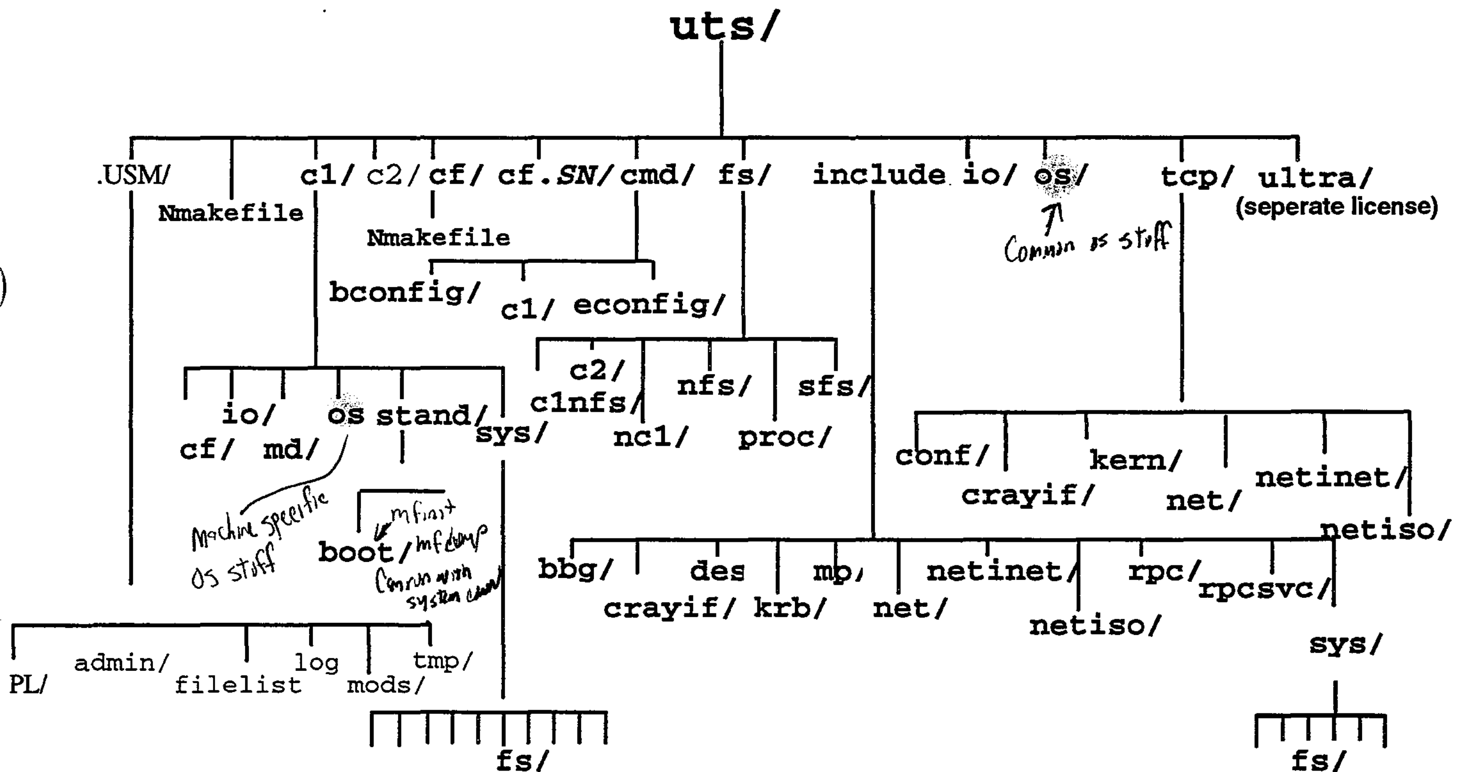


UNIX Time-sharing system uts/ (Kernel source code)

`.USM/`		UNICOS source maintenance directory
	`PL/`	Directories of kernel source in update-compatible program library format
	`admin/`	Files `srctree` and `version`
	`filelist`	List of pathnames and modifications under `uts/` directory
	`log`	Log of modifications applied
	`mods/`	Actual USM modifications applied to the program lists (PLs)
	`tmp/`	Scratch directory for USM
`Nmake-file`		Along with the compiled `Nmakefile.mo` and `Nmakefile.ms` makes up `cf.SN` directory and copies `cf/Nmakefile` to it
`c1/`		Cray-1 (CRAY Y-MP and CRAY X-MP specific source code)
	`cf/`	Configuration files (especially `conf.SN.c`: mainframe configuration file (by serial number) for disks, BMX/tapes, channels, and networks)
	`io/`	I/O drivers (except those common with Cray-2 systems)
	`md/`	Machine-dependent codes that includes all Cray assembly language (CAL), machine-dependent and low-level I/O routines, mainframe initialization routines, interrupt handlers, and low memory data
	`os/`	Operating system calls (includes sysent.c file) for system accounting, clock handling, process scheduling, memory allocation, and high-level I/O
	`stand/`	Stand alone kernel

	sys/	Cray-1 specific .h files such as structure definitions, C macros, and configuration information
c2/		Cray-2 specific source code
cf/		Contains the `Nmakefile` to build the kernel (copied to `cf.SN/`)
cf.SN/		Directory created by `uts/Nmakefile` to build a kernel for this machine; contains unicos binary and map
cmd/		Source code for kernel configuration command
	bconfig/	Source code for parameter file checker
	c1/kcom-press	Source code for kernel compression utility
	c1/mku-text	Source code for make utext.s command
fs/		File system algorithms
	c1nfs/	NFS file system - Cray to Cray only
	c2/	Cray-2 file system
	nc1/	New (current) Cray-1 file system
	nfs/	NFS file system - generic systems
	proc/	`/proc` file system
	sfs/	shared file system
include/		Header files
	bbg/	`Bus Based Gateway header files`
	crayif/	Hyperchannel structures
	des/	Data encryption structures
	krb/	Kerberos structures
	mp/	`mp.h`
	net/	TCP networking structures
	netinet/	TCP and UDP structures
	netiso/	ISO structures
	rpc/	Remote Procedure Call structures
	rpcsvc/	RCP strucures
	sys/	`c1/c2` common structures (includes `param.h`)
io/		Drivers common to Cray-1/Cray-2 (Ultra)
os/		System calls and other code common to Cray-1/Cray-2
tcp/		Kernel-level TCP source code

	conf/	*ioconf.c*
	crayif/	if_hy.c (IP / network driver link)
	kern/	
	net/	
	netinet/	
	netiso/	
[ultra/]		Ultranet driver source (separate license)



Object code organization

The figure on the right shows the placement and relative size of the object modules in a kernel binary configured for sn1920 (built on 6/92). It also illustrates which source code was used to generate each part of the kernel.

Tables (`lowmem.c`) and common blocks:

This is a large configuration that include 600 procs and 2200 inodes. Each memory resident, non-stack resident variable becomes a common block.

File system code:

NC1 and proc code become `lib/fs.a` in the kernel `make` procedure. NFS code is farther down in memory.

I/O drivers:

Most Cray block and character device drivers become `lib/io.a` in the kernel make procedure. Optional drivers are: `hsx` driver becomes `lib/hsx.a` (IOS Model D), `bmx` is high speed tape, `ipi3` is a customer ipi3 tape driver, and ultra is a Ultranet high-speed communications driver.

Machine-dependent code:

Machine-dependent code becomes `lib/md.a` in the kernel `make` procedure.

Operating system code:

`c1/os` and `uts/os` modules become `lib/os.a` in the kernel `make` procedure.

Security code:

If the system is non-secure, `nslog`, `nslogext`, and `nsecure` are included in `lib/nsec.a`. If the system is secure, `slog`, `slogext`, and `secure` are included in lib/sec.a.

TCP/IP code:

`uts/tcp` modules become `lib/tcp.a` in the kernel `make` procedure.

ISO code:

If ISO is configured, `uts/tcp/netiso` modules become `lib/iso.a` (else `lib/isostub.a` is made).

NFS code:

If NFS is configured, `fs/nfs` modules become `lib/nfs.a` in the kernel `make` procedure.

Startup code:

Codes used during startup are compiled into `lib/last.a` and linked to the end of the kernel. The symbol `pdummy` marks the part of this area which is to be overlaid by tables during startup. The symbol `binend` marks the end of the bootstrap kernel, but is meaningless after startup.

Object Code Organization

Not Updated 8.0

Contents		
Tables (lowmem.c and all "common blocks") Pointers to Tables (lowmem.c) Configuration variables (lowmem.c)		68.2%
~~fs/c1~~ fs/nc1 fs/proc ~~fs/c1~~	1.2% 1.7% .4% .6%	3.9%
c1/io		4.8%
c1/io/hx* c1/io/ebmx* c1/io/ipi*	0% 2.6% 0%	2.6%
uts/ultra/* [or io/unet_stubs]		2.3%
c1/md CAL *(1.4% of kernel code)* C	 .4% .6%	1%
c1/os uts/os		5.9%
io/[n]slog* os/[n]sec*		.1%
uts/tcp [uts/tcp/*iso]		4.8%
fs/nfs		4.9%
c1/md/init.c c1/md/csl* c1/md/pdummy.c c1/md/binend.s		.8%
		100%

* *optional: machine, site, or configuration dependent* (04002000 words)

Kernel logical organization

The figure on the right shows major logical divisions of UNICOS and the connections between them. Note the layered organization of the source code relevant to this block diagram.

User memory:

The `init` daemon, the idle processes, `esdpulse`, `utility` and other system daemon processes are vital to the operation of the system, but reside outside of the kernel and are executed in user (non-monitor) mode. The `init` and `inetd` daemons create (fork) the daemons and all the login shells.

System calls are made from all processes by loading the `S0` register with a call number, S1 with a calling argument address, and executing the `ex` instruction.

I/O channels:

The I/O channels are an example of an external cause of an interrupt to a CPU. On any interrupt to a CPU executing user code, an exchange takes place. The exchange operation loads the CPU's `P` register with the address of the single entry point into the kernel.

Entry to the `md/` layer:

The CPU enters the module `slave.s`, which analyzes the reason for the interrupt and jumps to one of the 15 handlers in the module `master.s`. Shown in brief detail are PCI (programmable clock interrupt), NEX (normal exit), and IOI (I/O interrupt).

System call layer:

There are 205 routines here that can be called through the `sysent[]` table. These routines include generic UNIX calls (they accept standard UNIX arguments and return standard UNIX values). In some cases, system calls are stand-alone processes; in other cases, they interface with the process control subsystem; and in some cases, they transfer data through the file-system management subsystem.

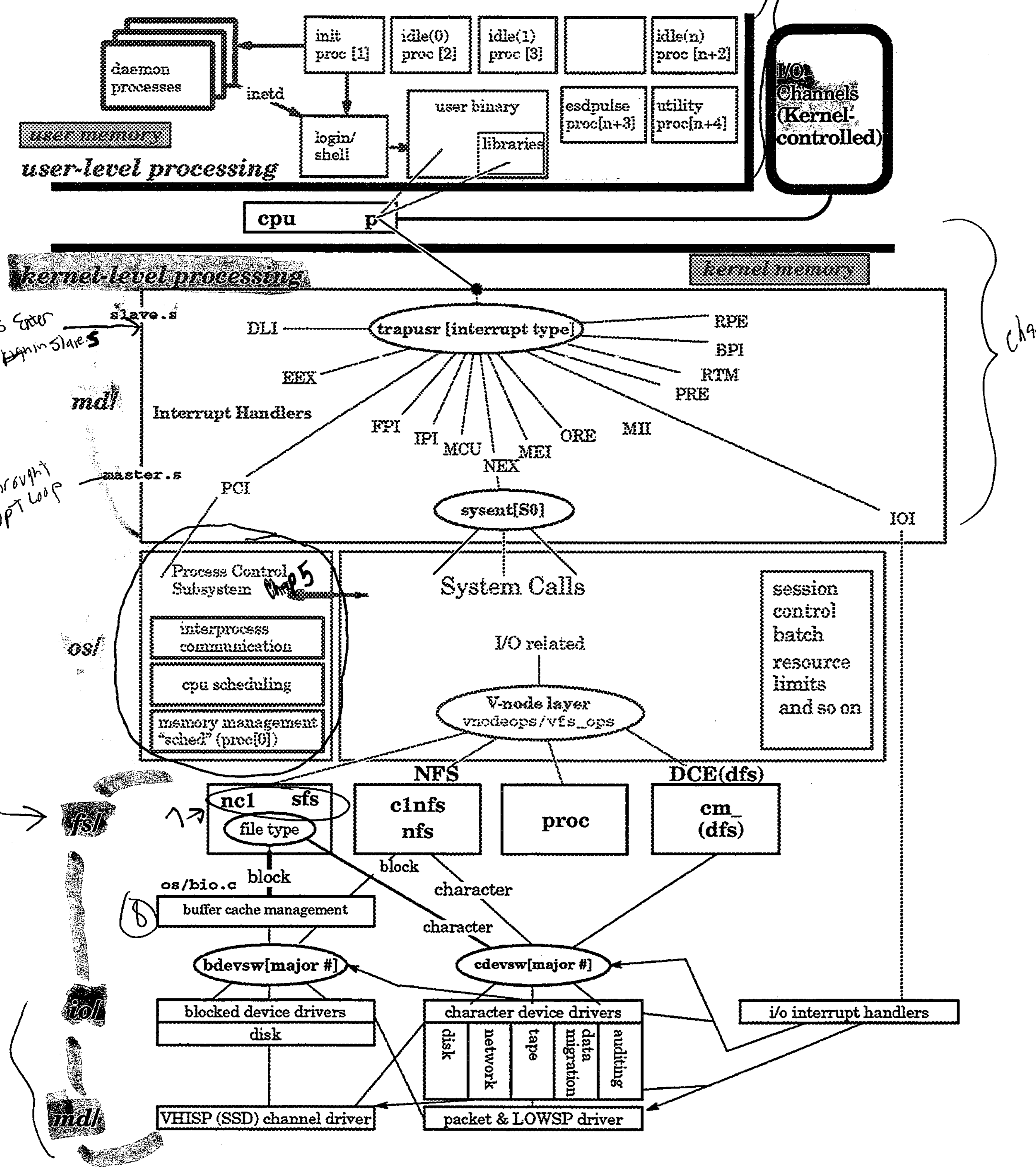
Process control subsystem:

These routines are not a separate layer, but provide miscellaneous service. One major service, process scheduling, is entered regularly through the handling of the PCI interrupt. Memory management is also frequently entered on a timed basis. The memory manager is not entered directly but is represented by entry 0 in the process table and is scheduled just like user process.

File system management layer:

Every file in UNICOS is represented by a device-independent vnode structure whether it resides on disk or any other medium. The vnode provides a common method of entry to file management. Vnode architecture has two jump tables `vnodeops` and `vfs_ops` which are used for vnode operations and virtual file system operations, respectively. UNICOS supports the UNICOS native file system (NC1 & SFS), optional network file systems C1NFS (optimised for Cray to Cray) and NFS, /proc file system, and optional Distributed File System (under DCE) DFS. The NC1 files can be either blocked or character type. The NFS routines use the RPC/UDP/IP route to the network driver (a character device). The `proc` routines normally do not use a driver; they read and write from process memory.

Kernel Logical Organization



Buffer cache management:

The NC1 file system is read and written through system buffers (unless the file is opened for raw I/O). In any case, these routines must search the buffers for any blocks they are accessing. The buffer management routines are in module `os/bio.c`, for historical reasons.

Blocked device drivers:

There are 12 block device drivers (IOS B/C/D and E versions). They include DD*nn* disks, IOS expander disk, Buffer Memory, SSD, Ldcache, Striped and Central Memory (RAM) devices.

Character device drivers:

There are over 50 character drivers (IOS B/C/D and E versions) allowing users access to all kinds of real I/O devices and other system resources, all by means of character special inodes.

I/O interrupt handlers:

The interrupt handlers are the back ends of the drivers, entered on the IOI interrupt at channel completion.

VHISP channel driver:

These are the CAL routines called by the block drivers and interrupt handlers to queue and dequeue requests. These routines actually provide the functioning the SSD channels.

Packet and LOWSP channel driver:

These are the routines called by the block drivers, character drivers, and interrupt handlers to queue and dequeue requests for the IOS and actually provides the functioning the low-speed channels.

This page used for alignment

Kernel structures for process control

The figure on the right shows the major kernel structures used for controlling the execution of a process in the system. In “Hardware” illustration,“CPU N” is an abbreviated version of the CPU data and control registers. “Clusters” shows hardware cluster ”2–1” used by user processes. Hardware cluster “1” is always used by CPU’s executing the kernel.

Process working storage table, or PWS”:

- Has one entry for each CPU on the system.
- Each PWS entry contains information about a specific CPU and is used to control it; much of the control information is used for context switching.
- The PWS table must be assigned to locations 0 through 4095 (decimal) in kernel memory because the table entry assigned to a CPU contains hardware exchange packages, and hardware exchanges can only occur in locations 0 through 4095 (decimal) in low memory.
- Contains pointers that “link” this CPU to its currently connected process.

Process table, or ”proc table”:

- Focal point of control for all processes in the system.
- Contains one entry for each process that could potentially exist in the system.
- A typical `proc` table is configured with approximately 400 entries.
- Because the `proc` table is always resident in memory, the information maintained in the `proc` table entry for a particular process is always available to the kernel, even when the process is swapped out of main memory.

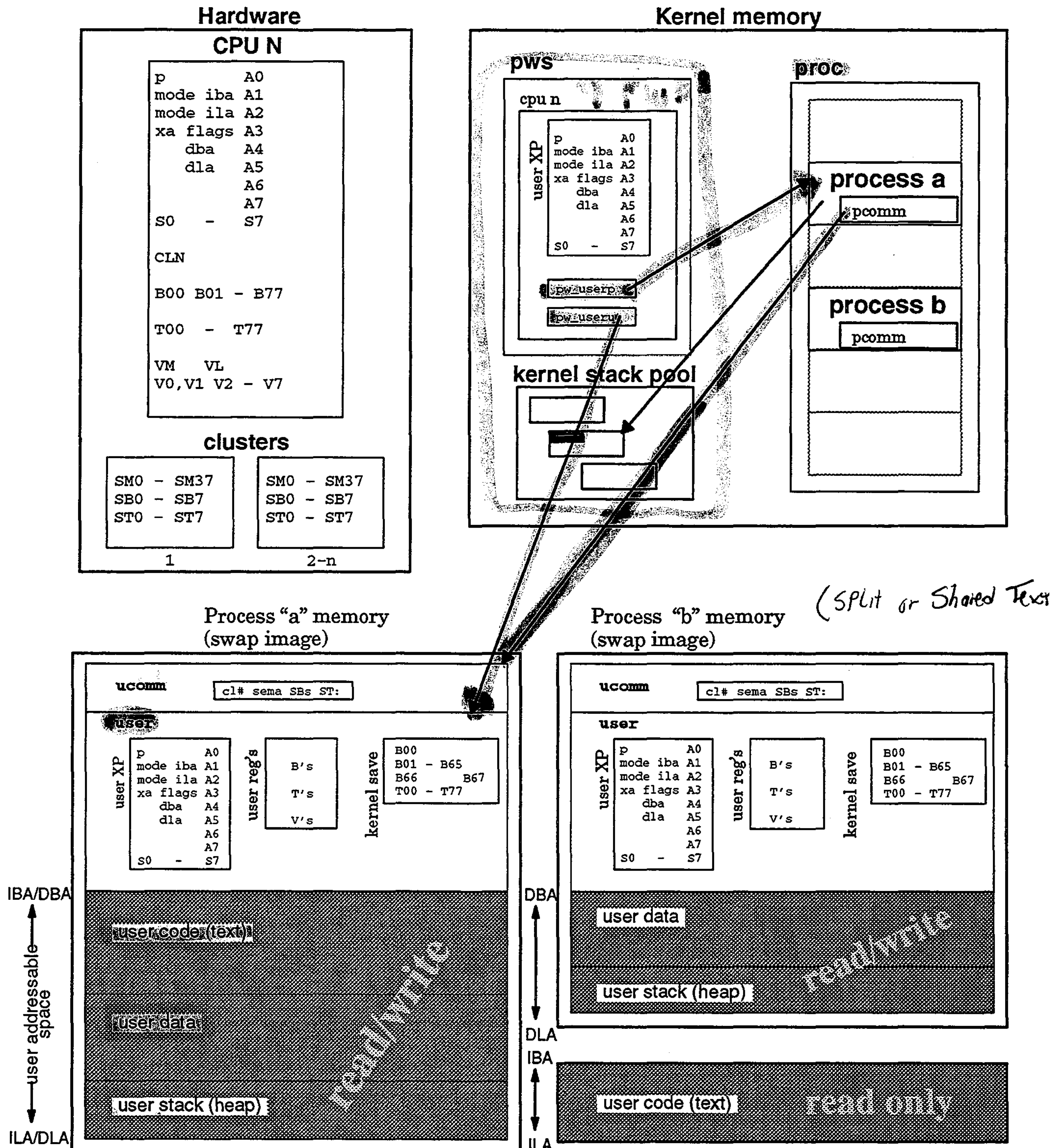
Process common area, or ”pcomm area”:

- The process common area (`pcomm`) is a substructure of a process’s `proc` table entry. It contains additional kernel process management fields.
- The kernel uses the `pcomm` area to manage both multitasked and non-multitasked processes.
- Members of a multitasking group (siblings) run in parallel and share a common memory space, therefore, the kernel memory management fields reside only in the `pcomm` area of the `proc` table of the oldest (first) sibling.
- The `pcomm` areas in the `proc` areas of the younger siblings are unused. A field in each younger sibling `pcomm` points to shared `pcomm` area of the oldest sibling.

Kernel Stack Pool:

- The kernel assigns a stack area from this pool for each active process in the system.
- The dynamically allocated area is in upper memory (above user processes).
- A pointer in the process table locates the stack area for a given process.

Kernel Structures for Process Control



The figure shows a simplified picture of two user memory images. The ucomm and user areas in a memory image are part of a user's process but cannot be directly addressed by the user. The ucomm and user areas are used by the kernel to manage the user's process and are only accessible to the system.

User area:

- Used in context switching to save and restore user information.
- Contains the kernel's register save areas.

User common area:

- Contains information relating to memory control and information common to multiple processes in a multitasking group.

Combined and split text :

The simplified picture of user process *A*'s memory image shows where user code (text) and data are combined and loaded into one process image. User process *B*'s memory image shows where the user code is split off from the data and loaded as a separate element in the system. When a program is compiled and loaded, the user can make a request to the loader (`-n` option to `segldr`) to split off the user code from the data and share it with another process.

The figure shows the user addressable space for process *B*'s memory image defined by the instruction and data base and bound limits. In the case of a regular process, the user process has read and write access to the entire user addressable area.

The figure also shows the user addressable space for split text. The data area defined by the DBA/DLA is read/write addressable. The shared text segment defined by the IBA/ILA is read addressable only. This prevents the shared text segment from being written to and altered by one of the sharing processes.

Multi-tasked group :

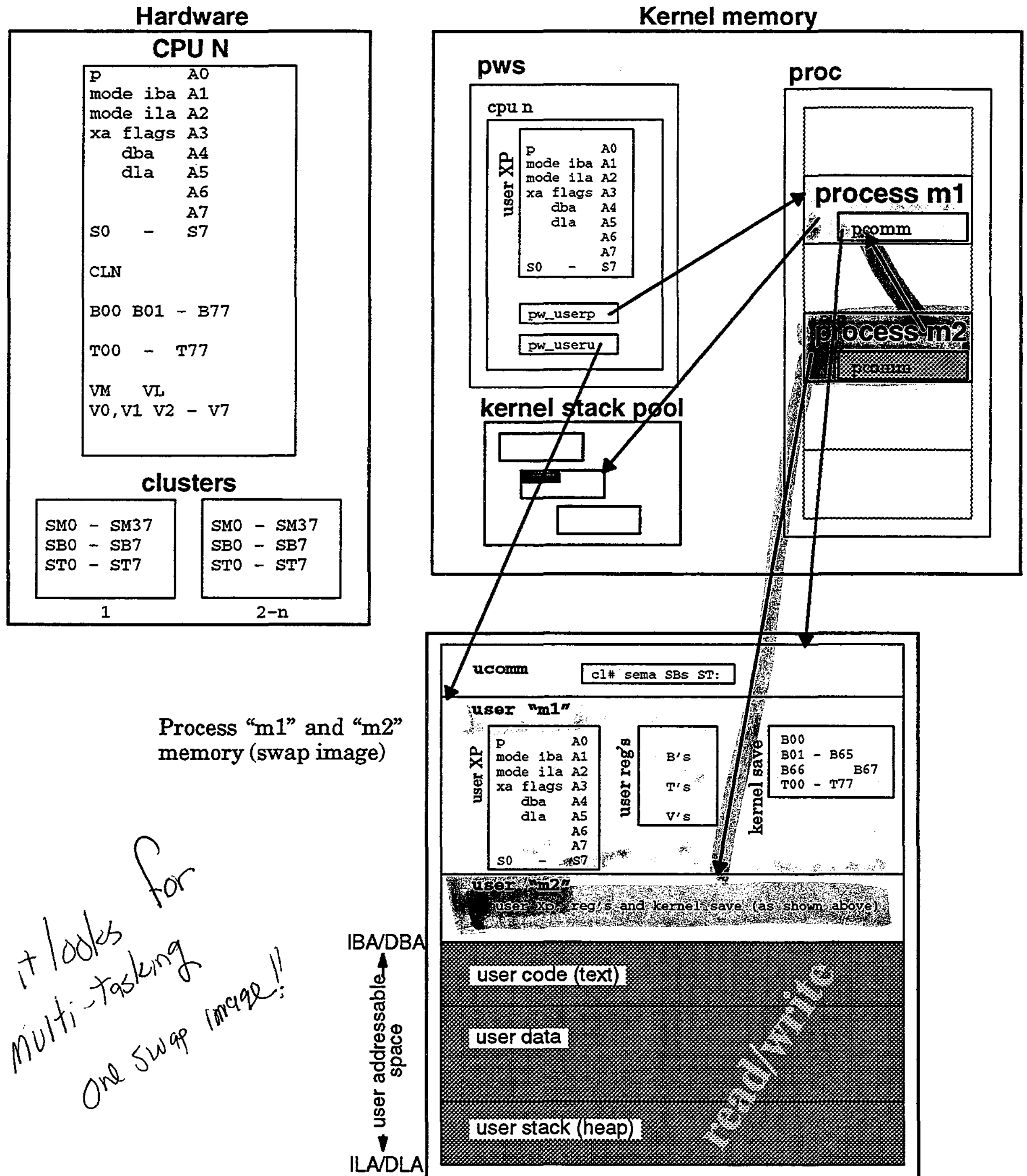
Processes "m1" and "m2" illustrate a multi-tasked group of processes. Members of an m.t. group are called "siblings" (not the usual parent / child).

Each member of the group has its own `proc` table entry associated with a user area and can be independently connected to different CPUs (at different or the same time). Only the eldest sibling has the `pcomm` data, the younger sibling proc entries point to the eldest sibling's. Note that each sibling's proc entry has an associated `user` area, but only the shared eldest sibling's `pcomm` has a `ucomm` associated with it.

The swap image of the m.t. group contains one shared `ucomm` area, unique `user` areas, and one shared addressable program image. Each member of the group may be individually connected to a CPU, and executing the user program and/or system (kernel) code.

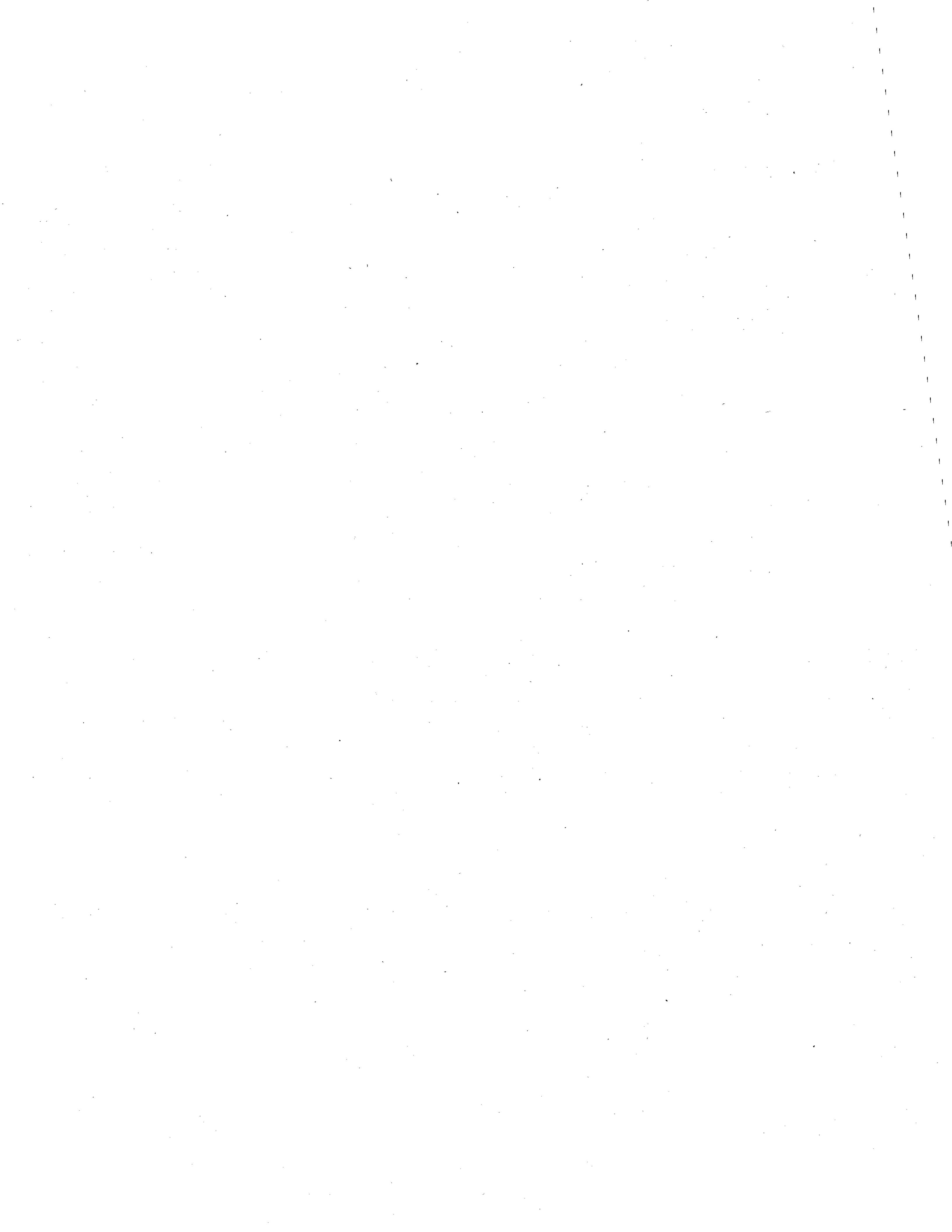
The illustration shows an m.t. group with combined text and data. A "split" text memory image (not shown) is valid for an m.t. group as well, following the same format as shown on the previous page,

Kernel Structures for Process Control



The figure shows a simplified picture of a multi-tasked group of processes. Each member of the group shares the same code (text) and data.

This page used for alignment



Contents

Hardware Overview [2] 2–1
- Objectives 2–1
- Hardware Identification by serial number 2–2
- CRAY Y-MP System with IOS model D 2–4
- CRAY Y-MP System with IOS model E 2–6
- IOS model E 2–8
  - I/O overview 2–8
  - General packet format 2–10
  - Packet types 2–10
  - Disk request packet 2–12
  - Simple disk read 2–12
  - Write behind 2–14
  - No write behind 2–14
- CRAY Y-MP EL differences summary 2–16
- CRAY Y-MP system control 2–19
  - Registers in each CPU 2–20
  - Shared resources 2–30
- CRAY Y-MP C90 system control 2–32
  - Registers in each CPU 2–32
  - Status registers 2–34
  - Shared resources 2–34
- CRAY X-MP system control 2–35
  - Registers in each CPU 2–35
  - Shared resources 2–35
- Interrupt processing summary 2–36
- Exchange (XP information) 2–40
  - Exchange package `crash` display 2–40
  - XP mode and status bit breakdown 2–40
- Memory addressing modes 2–42
  - CRAY-1 and CRAY X-MP without EMA 2–42
  - CRAY X-MP in compatibility modes 2–44
  - CRAY X-MP EMA instruction formats 2–47
  - CRAY Y-MP compatibility (24-bit) mode 2–49
  - CRAY Y-MP EAM (32-bit) mode 2–51
  - CRAY Y-MP EAM instruction formats 2–53
- Summary of hardware types / binary restrictions 2–55
  - CRAY Y-MP C90 in native mode 2–57
  - CRAY Y-MP C90 in compatibility mode 2–57
  - CRAY X-MP in compatibility mode 2–57
- Hardware system control foldouts 2–59
  - CRAY Y-MP system control 2–61
  - CRAY Y-MP C90 system control 2–62
  - CRAY X-MP system control 2–63

Hardware Overview [2]

Objectives

After completing this section you should be able to:

- Survey the hardware environment of the following systems:

  CRAY Y-MP / IOS-D

  CRAY-MP / IOS-E

  CRAY Y-MP EL

- Summarize the function of the IOS in relation to UNICOS
- Survey the mainframe architecture of the CRAY X-MP and CRAY Y-MP and define the role of each component in relation to UNICOS
- Describe the interrupt exchange sequence and kernel interrupt processing in general
- Document the exchange package format with sample crash(8) output
- Provide a detailed description of the various addressing modes on the CRAY X-MP and CRAY Y-MP systems

Hardware identification by serial number

MFTYPE		Series	# of CPUs	MFSUBTYPE
CRAY1A	CRAY-1 (A,B,S)	sn1 – sn57		CRAY1XX
CRAY1S	CRAY-1 M	M1 – M9		CRAY1XX
CRAY1M				
CRAY_2	CRAY-2	Q1, Q2, sn2001 – sn2029, 2101	8	
CRAYXMP	CRAY X-MP	100	2	XMP1XX
		200	2,4	XMP2XX
		300	1	XMP3XX
		400	1,2	XMP4XX
		500	1 (14se)	XMP5XX
		600		XMP6XX
		1100	2,4 (EA)	YMP1XX
		1200	1,2 (EA)	YMP2XX
		1300	1 (EAse)	YMP3XX
CRAYYMP	CRAY Y-MP			
		1000	8 (max.)	YMP0XX
		1400	2 (max.)	YMP4XX
		1500	4 (max.)	YMP5XX
		1600	1,2 (2E)	YMP6XX
		1700	8I (2–8)	YMP7XX
		1800	8E (4–8)	YMP0XX
		1900	4E (2–4)	YMP9XX
		2400	M90 (4) (DRAM)	YMP11XX
		2600	M90 (2) (DRAM)	YMP12XX
		2800	M90 (8) (DRAM)	YMP10XX
MFTYPE		**Series**	**# of CPUs**	**MFSUBTYPE**
CRAYC90	CRAY Y-MP C90	4000	16	C900XX
CRAYXMS				
CRAYEL	CRAY Y-MP EL	5100	1–4	XMP6XX

The following IOS models are identified by serial number:

MFTYPE	Series
IOS-A	sn3 – sn5, sn7 – sn9
IOS-B	sn6, sn11 – sn61, sn63 – sn89
IOS-C	sn62, sn101 – sn191
IOS-D	sn401 – sn499
IOS-E	sn701 – sn799

(If the IOS model is not stand-alone, it has no serial number of its own)

This page used for alignment

CRAY Y-MP system with IOS model D

Mainframe

The Y-MP has more low-speed channels, allowing connections to 8 IOP's

SSD

VHISP channel numbers to the SSD were changed from those of the X-MP (6/7) to 1/5.

IOS Model B/C

- **Buffer Memory:**

  Same as models B and C. As with model C, each IOP has 2 HISPs (to buffer memory and central memory) that can be linked together so that transfers between buffer memory and central memory bypass IOP local memory (Memory Bypass I/O instructions).

- **Master I/O Processor (MIOP):**

  The MIOP has the Master Clear and MCU interrupt capability to the mainframe, but no longer need be "master" over the other IOPs. Accumulator channels still exist but are not so important since UNICOS 5.0.

  Each IOP can have its own low-speed pair and HISP to the mainframe.

  With an Operator Work Station, **operator control** is by means of a windowed large-screen color monitor. Operator interfaces with the IOP kernels also appear as windows on the same monitor. Remote operation is possible where the OWS is connected to a network. The OWS provides a direct operator interface to the mainframe kernel via special packet types passed through the IOS between the OWS and UNICOS. Also, the OWS functions as a normal UNIX front-end, allowing regular TCP/IP communication between operators and UNICOS.

  Deadstart and maintenance storage is on the OWS and MWS. Deadstart of the IOS is done via OWS software, which can send a hardware master clear signal on a wire in its channel cable to the MIOP. Hardware errors are logged by the MWS. The MWS is cabled to the mainframe, SSD, or IOS for offline diagnostics.

  The MIOP still handles all **network** I/O, with the added efficiency of having its own HISP to central memory.

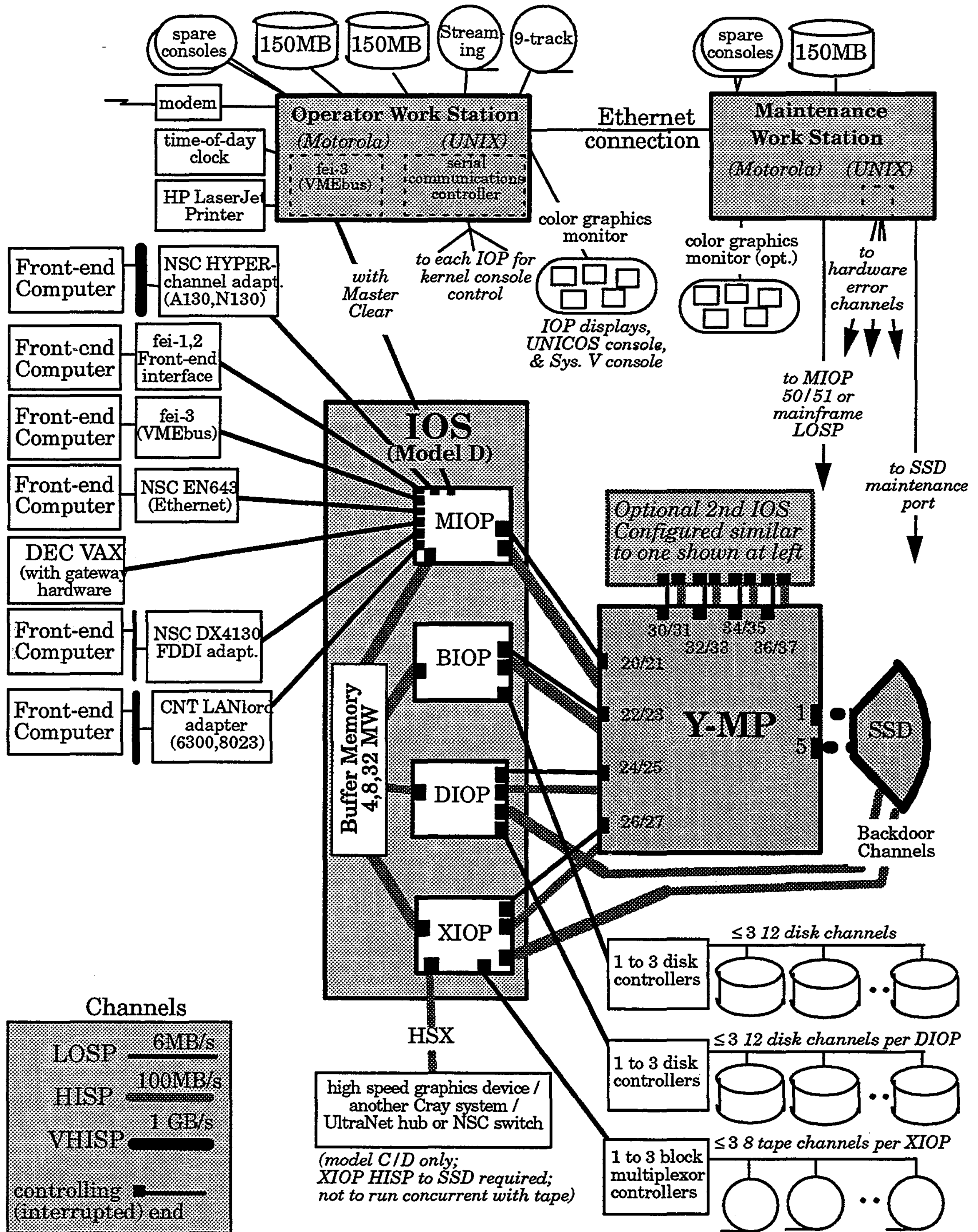
- **Buffer I/O Processor (BIOP) and Disk I/O Processor (DIOP):**

  Same as models B and C except one DMA port is reserved for a HISP to SSD memory, so only 3 disk controllers are normally attached to each IOP.

- **Auxiliary I/O Processor (XIOP):**

  Basically the same as for models B and C. The XIOP is still preferred for driving the backdoor channel to the SSD, though another IOP such as the DIOP could be used. Up to 8 block multiplexor channels connected to IBM compatible tape devices can be attached to each XIOP.

CRAY Y-MP System with IOS Model D



CRAY Y-MP system with IOS model E

Mainframe

Same as above; configurable memory increased.

IOS Model E

See *I/O Subsystem Model E (IOS-E) Support Guide*, SD-2107, for more information.

- **Buffer memory:**

  Buffer memory does not exist on the IOS-E. With no shared buffer memory and no accumulator channels, each I/O Cluster (IOC) is independent of the others. The lack of shared buffer memory requires all software to reside in the 64K-parcel EIOP or MUXIOP local memory. Disk I/O error correction code has moved to the mainframe driver.

- **Operator control / deadstart:**

  The OWS and MWS for the IOS-E are Sun workstations. They run Cray Research OWS-E and MWS-E software. Each workstation is connected to the Service Workstation Interface (**SWI**) module. The SWI fans the connection out to every IOP (MUXIOP and EIOP) of every cluster in the IOS. Either workstation can **master clear** the entire IOS, one cluster, or one IOP. They can also transfer data to/from any IOP and do diagnostic functions. The OWS **operator interface** with UNICOS is through this channel, with zip making the OWS console a terminal to UNICOS. TCP/IP is not supported without another channel to a CCA.

- **I/O cluster (IOC):**

  An IOS-E is a set of independent IOCs. Each I/O cluster is a MUXIOP connected to 4 EIOPs. Clusters are only interconnected through their common connections to work stations, mainframe, and SSD. EIOP software is written in the ELAN language, rather than APML.

- **MUXIOP:**

  The MUXIOP controls the packet (low-speed) channel pair, the HISP to central memory and, if present, the backdoor HISP to the SSD.

- **EIOP:**

  Each Type E IOP receives requests from its MUXIOP and drives its devices through its 4 channel adapters. The 4 adapters must be of the same type (see CCA, HCA, DCA, and TCA below) because of the EIOP's limited size (64K 16-bit parcel) for software.

- **Channel adapters / buffers:**

  Each channel adapter reads and writes from/to its own 64K 64-bit word circular buffer. Each adapter can be connected to only 1 device. Two channels to the same adapter is only possible with an fei-4. The following adapters are used to make station connections:

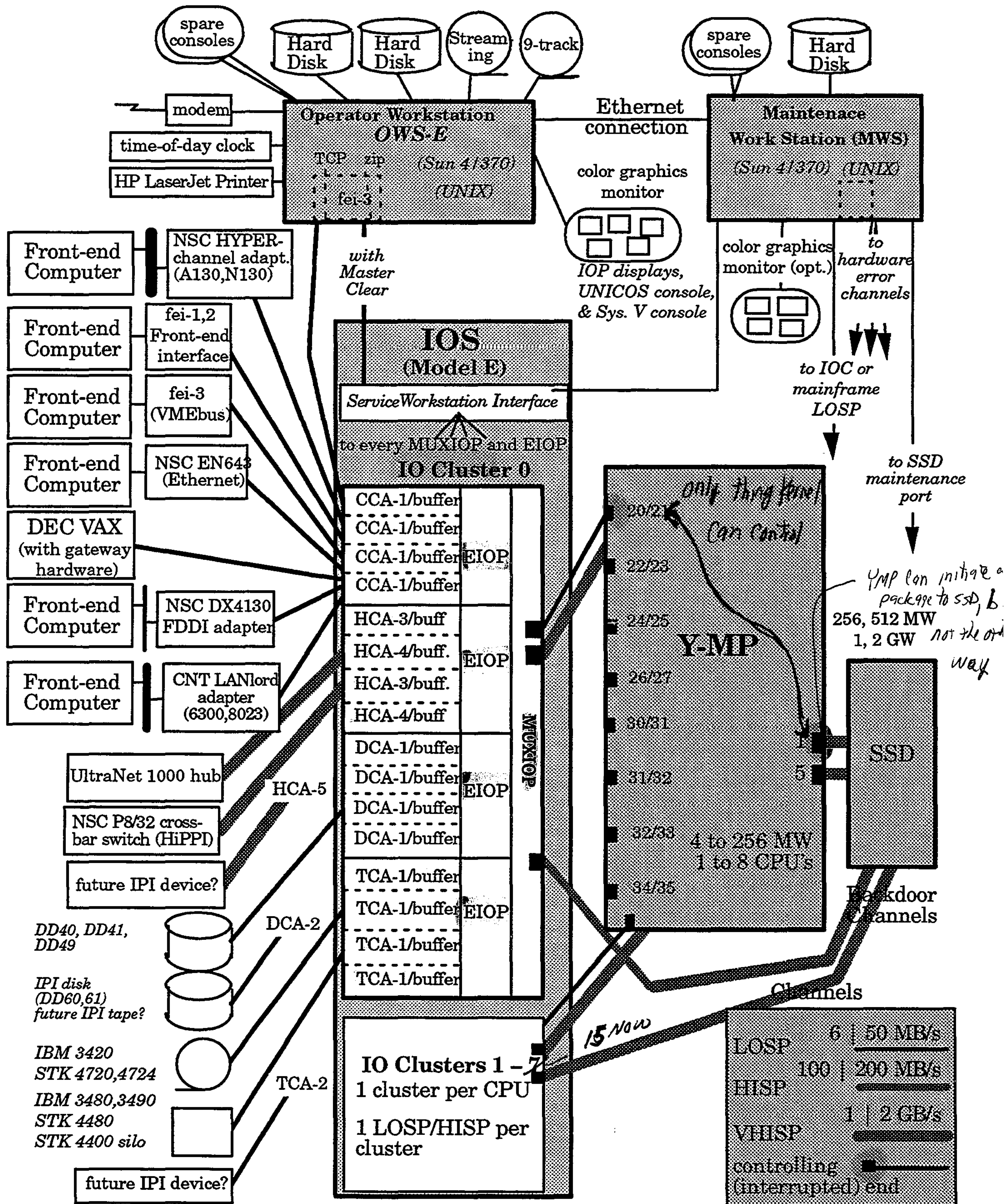
  Low-speed (6 or 12 M Bps) communications adapters (**CCA-1**)

  High-speed (100 M Bps) channel communications adapters (**HCA**)

  Disk channel adapters (**DCA**)

  Tape channel adapters (**TCA**)

CRAY Y-MP System with IOS Model E



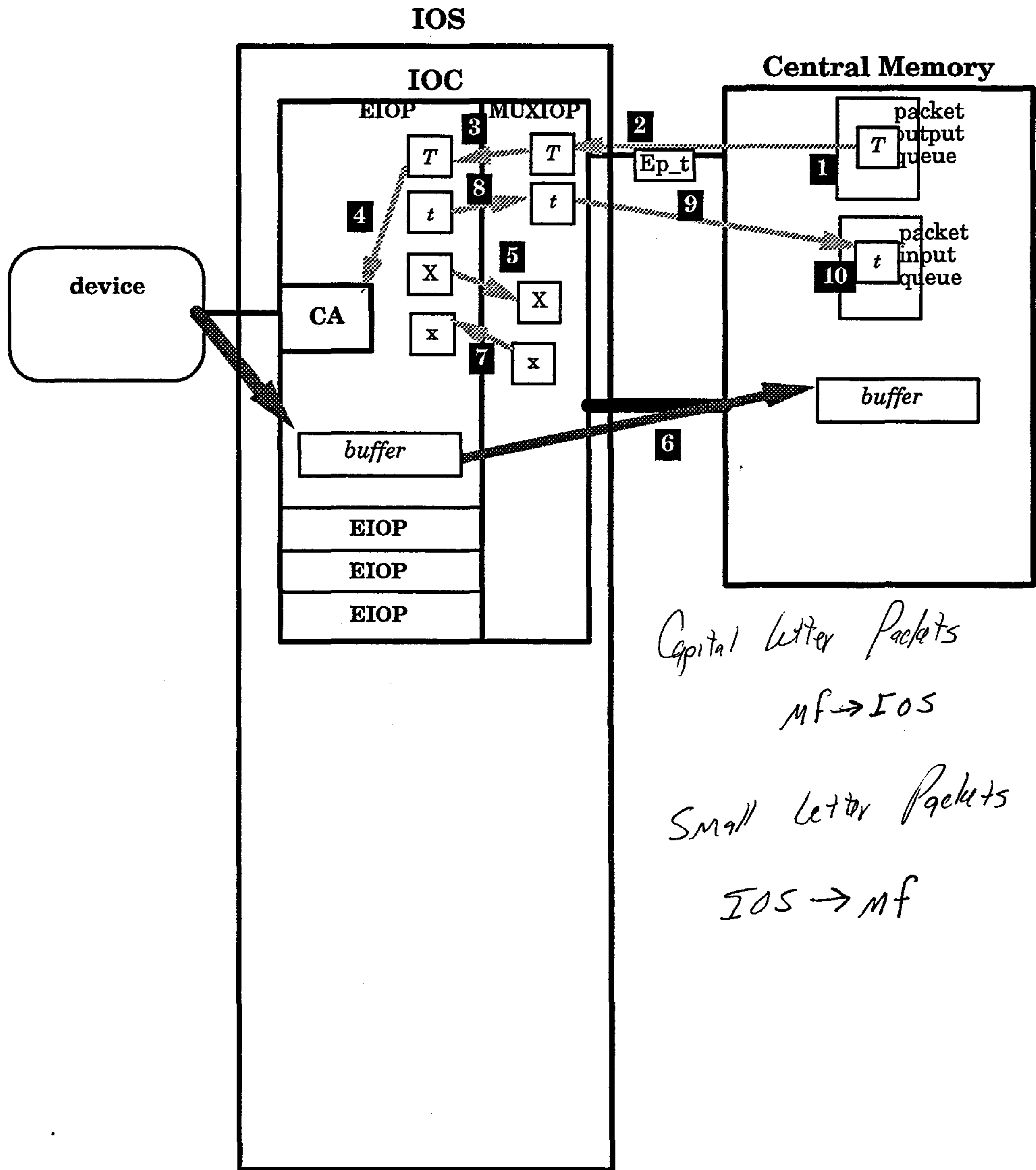
IOS model E: I/O overview

The diagram on the right shows channel use and IOS packet processing by illustrating a sample UNICOS read directed to a Model E IOS cluster

1. An IOS packet of type "*T*" (shown on following pages) is created by the kernel device driver and queued to be written to the IOS in the packet output queue. For most packet types the packet contains kernel I/O request information only, no data.
2. The packet is written to the appropriate IOS cluster (IOC). The packet is delivered via the low-speed channel when the IOS executes a corresponding read on the same channel. When the LOWSP packet transfer completes the mainframe gets on output interrupt, the IOS gets an input interrupt.
   a. The mainframe checks if any other packets are queued waiting transfer, if so it initiates that packet's transfer to the IOS.
   b. The mainframe is free to go about other work but should not use the buffer's data until the I/O is complete.
   c. The IOS proceeds to perform the request indicated by the packet.
3. The MUXIOP writes the packet to the EIOP which handles this device based on packet header information (packet type).
4. The EIOP commands the channel adaptor (CA) for the corresponding device to read the data into buffer areas reserved within the EIOP.
5. The EIOP instructs the MUXIOP to deliver the data via the high-speed channel to the central memory address provided in the original request packet.
6. The data is transferred into the waiting central memory buffer. *Note: data may be transferred to an SSD memory address using a backdoor high-speed channel under the control of a IOS packet "target memory" field*
7. The MUXIOP informs the EIOP when the data transfer is complete.
8. The EIOP constructs a response packet type "*t*" and sends it to the MUXIOP indicating the I/O operation is complete. Fields in the packet show the completion status.
9. The MUXIOP writes the response packet to the mainframe via the low-speed channel. The packet is delivered when the mainframe performs a read on the corresponding channel. The packet is read into the packet input queue.
10. The receipt of the packet causes an input interrupt on the mainframe and an output interrupt on the IOS.
   a. The IOS proceeds to process other packet requests, this one is complete.
   b. The mainframe selects a new packet input area and initiates the next read (before the last one is processed as below).
   c. The mainframe selects and executes the correct packet's input interrupt handler.
   d. The incoming packet usually indicates the completion of an earlier I/O request. The error checking and final I/O processing is performed including informing user process that the request is complete.

IOS Model E: I/O Overview

IOS Channels and Packets



General packet format

The figure on the right shows the generic IOS E packet type **(Ep_t)**. Packet definitions are found in the header file `c1/sys/epack.h`.

- Epacket: header of structure Epacket (see Ep_trailer for contents).
- Ep_type: packet type indicating the type of request the packet represents.
  - Packets sent to the IOS use letters *A – Z*.
  - Packets returned by the IOS use letters *a – z* (for example, request *D* – response *d*).
  - Current packet types are shown below in Packet types.
- `Ep_data`: the packet body itself made up of a variable number of words depending on packet type. A sample disk packet is shown on following pages.
- `Ep_trailer`: the header information repeated again as a trailer for packet validation.
  - `Ep_magic`: used to validate packet contents.
  - `Ep_length`: length of packet in words (includes header and trailer).
  - `Ep_source`: shows where packet originated – always *13* for the mainframe.
  - `Ep_cluster`: IOS cluster number EIOC.
  - `Ep_proc`: IOS process number EIOP, destination for packet.
  - `Ep_flags`: used to indicate processing options.
  - `Ep_lpath`: driver dependent logical path for request – indicates device unit number.
  - `Ep_seq`: packet sequence number – used to identify the packet in the packet queues.
  - `Ep_ackseq`: packet acknowledge number indicates packets of this number and lower have been sent and validated.

Packet types

Packet types defined in **`c1/io/epackin.c`**.

```
/*  Packet source/destination ids for model E IOS    */
#define EPKT_DSK     'd' /*  Disk response    */
#define EPKT_HIPPI   'h' /*  High speed comm */
#define EPKT_IPI     'i' /*  IPI responses    */
#define EPKT_LSP     'n' /*  Low speed comm   */
#define EPKT_OWS     'o' /*  Operator workstation   */
#define EPKT_SYS     's' /*  System services */
#define EPKT_BMX     't' /*  Tapes        */
#define EPKT_ZTY     'z' /*  Z terminals      */
```

IOS Model E: I/O Overview

E packet format

Ep_t *(c1/sys/epack.h)*

Epacket *Epacket (c1/sys/epack.h)*		
Ep_type		
Ep_data [EPAK_MAXLEN-2] *(67)*		
Ep_trailer *Epacket (c1/sys/epack.h)*		

Field	Description
Ep_magic (6)	Magic number *(octal 55)*
Ep_length (10)	Length in words
Ep_fill (3)Filler	(unused)
Ep_source (4)	Location of code which sent packet EPAK_SRC_MF *(13)* EPAK_SRC_OWS *(14)* EPAK_SRC_MWS *(15)*
Ep_cluster (5)	IOS cluster number (EIOC)
Ep_proc (4)	IOS processor (EIOP)
Ep_flags (8)	Processing options
Ep_lpath (8)	Logical path of code to process request
Ep_seq (8)	Packet sequence number to validate
Ep_ackseq (8)	Sequence num ber of last valid packet

Disk request packet

A partial layout of a disk request packet is shown on the right. A full definition of all fields is found in header file `c1/sys/epackd.h`.

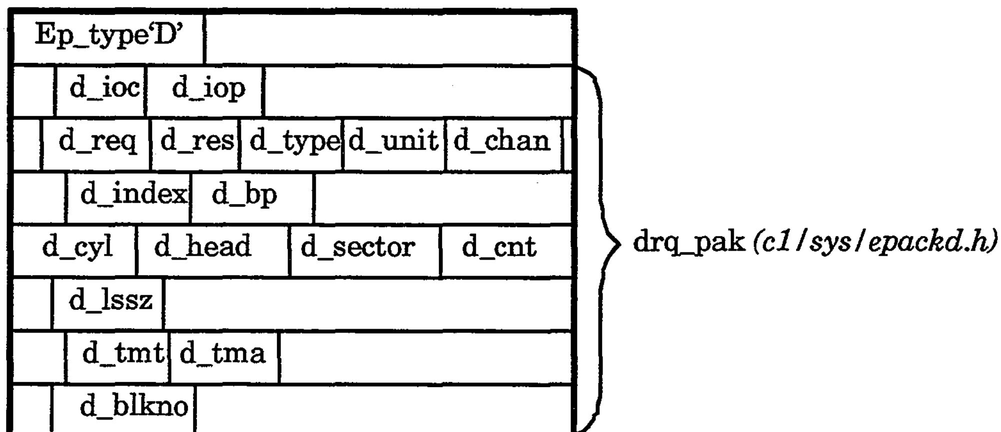
- `Ep_type`: packet type "D" indicated a disk packet.
- `drq_pak`: disk request packet body (Note: not all fields shown)
  - `d_ioc`: IOS cluster EIOC number
  - `d_iop`: IOS processor EIOP number
  - `d_req`: disk request code
  - `d_res`: disk response code
  - `d_type`: device type
  - `d_unit`: device unit number
  - `d_channel` number
  - `d_index`: index to device table entry (`pdd_tab`)
  - `d_bp`: pointer to request buffer
  - `d_cyl`: disk cylinder number
  - `d_head`: disk head number
  - `d_sector`: disk sector number
  - `d_cnt`: request length in sectors
  - `d_lssz`: logical sector size
  - `d_tmt`: target memory type (MF or SSD)
  - `d_tma`: target memory address
  - `d_blkno`: absolute block number (device relative)

Simple disk read

1. Mainframe creates a "D" packet with fields as shown above and queues a disk request in its packet output queue. The packet waits its turn, and is written on the low-speed channel to MUXIOP.
2. MUXOP receives the packet on the low-speed channel and routes it to the indicated EIOP.
3. EIOP gives the request to its disk driver software which issues the appropriate channel control functions to the DCA channel adapter to send data to the device.
4. Channel adapter translates to control signals for the disk. The disk reads data. Channel adapter assembles data into 64-bit words and writes it to its I/O buffer.
5. EIOP sends an X-packet to the MUXIOP to request a transfer to central memory.
6. MUXIOP functions the high speed channel to transfer the sector to central memory. No interrupt occurs on mainframe for this channel.
7. MUXIOP sends a transfer response (x-packet) to the EIOP.
8. EIOP creates a response (d-packet) to the original D-packet and sends it to MUXIOP.
9. MUXIOP writes the d-packet to mainframe. The interrupt caused by the packet read completion causes mainframe to see I/O completion.

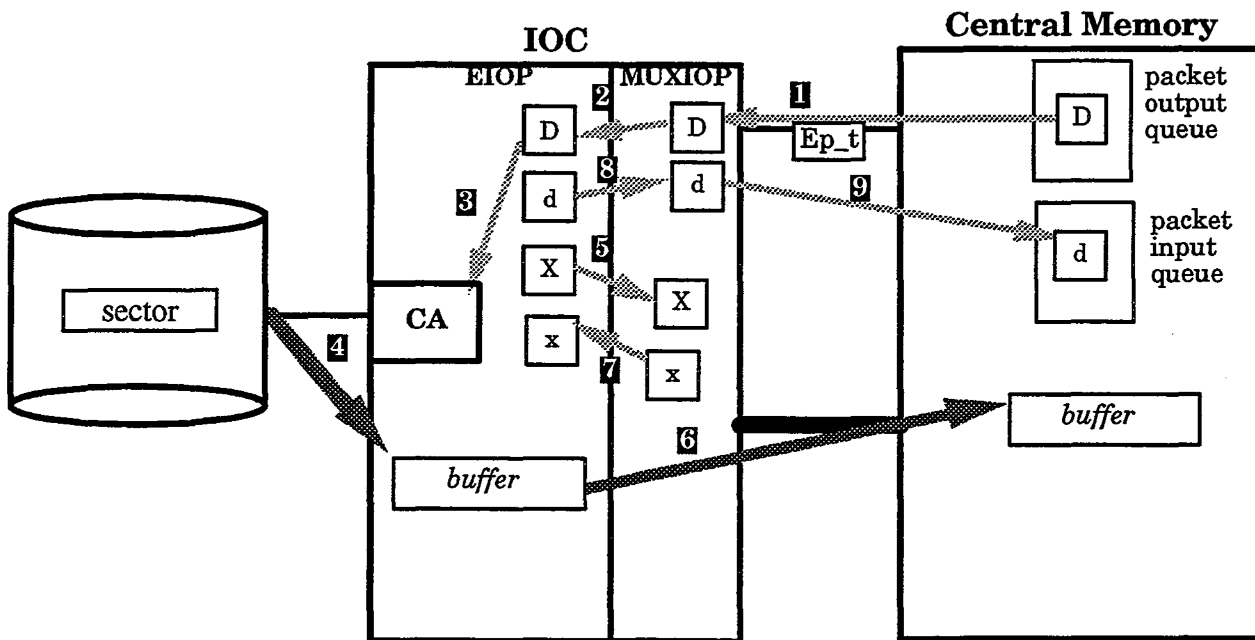
IOS Model E: I/O Overview

E disk request packet format



Fields in relative position only

Disk Read Example



Write behind

The default type of write is called "write behind". As shown below, the IOS reports the transfer is complete (returns packet) after obtaining data from central memory into IOS memory. The actual disk transfer is performed after this packet is returned.

1. Mainframe queues a disk request as a D-packet in its packet output queue. The packet waits its turn, and is written on the low speed channel to the MUXIOP
2. MUXOP receives the packet and routes it to the indicated EIOP.
3. EIOP sends an X-packet to the MUXIOP to request a transfer from central memory.
4. MUXIOP functions the high speed channel to transfer the sector from central memory to the I/O buffer. No interrupt occurs on the mainframe.
5. MUXIOP sends a transfer response (x-packet) to the EIOP.
6. EIOP creates a response (d-packet) to the original D-packet and sends it to MUXIOP.
7. MUXIOP writes d-packet to mainframe. The interrupt caused by the packet read completion causes mainframe to see I/O completion.
8. EIOP gives the request to its disk driver software. This software issues the appropriate channel functions to the DCA channel adapter to send to the device.
9. Channel adapter translates to control signals for the disk. The disk writes data.
10. If a disk error occurs another "d" packet is returned. The kernel disk driver will log the error and begin a retry process.

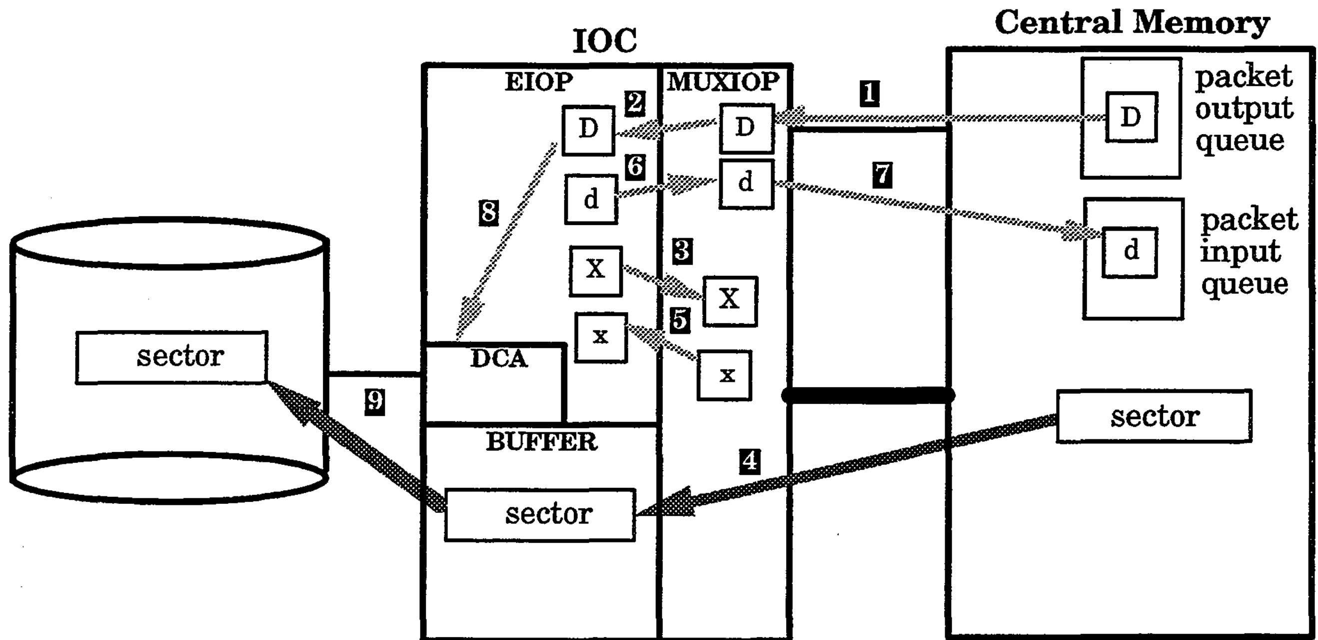
No write behind

A file can be opened for set to "no write behind" with the O_SYNC option (flag). As shown below the IOS sends the reply packet after the disk channel adapter has reported the write is complete. *For some disk models (e.g. dd40) the channel reports the write is complete before the data is actually on the disk platter.*

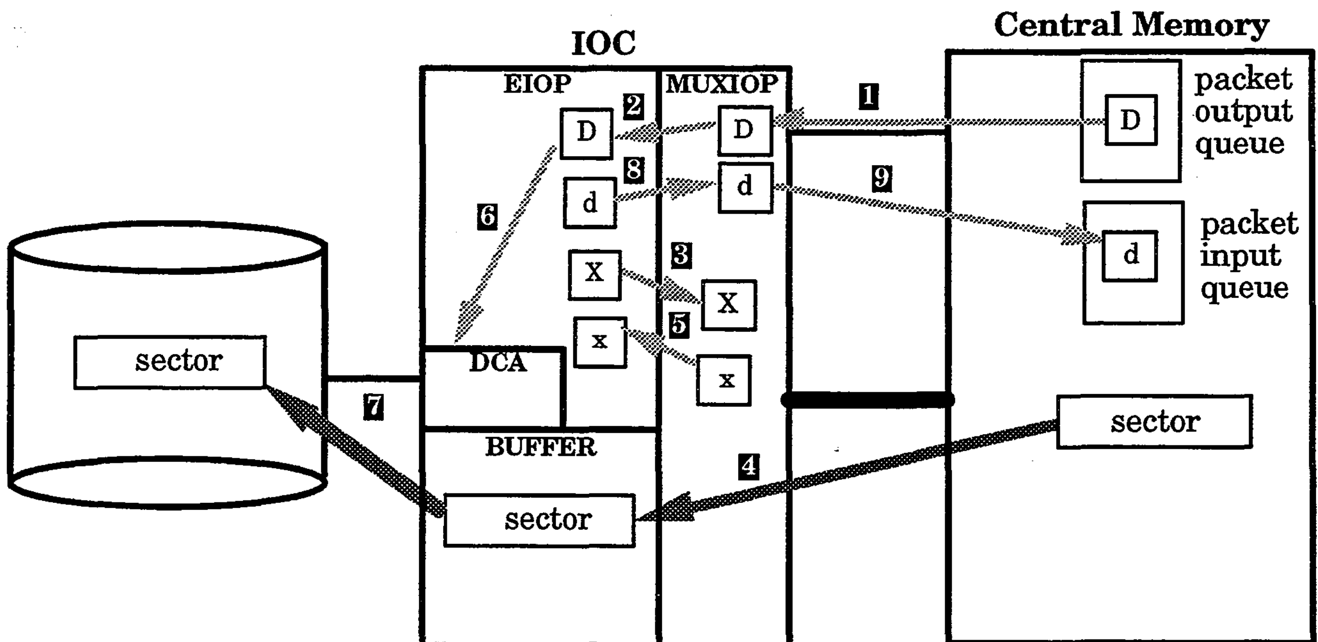
1. Mainframe queues a disk request as a D-packet in its packet output queue. The packet waits its turn, and is written on the low speed channel to the MUXIOP.
2. MUXOP receives the packet and routes it to the indicated EIOP.
3. EIOP sends an X-packet to the MUXIOP to request a transfer from central memory.
4. MUXIOP functions the high speed channel to transfer the sector from central memory to the I/O buffer. No interrupt occurs on the mainframe.
5. MUXIOP sends a transfer response (x-packet) to the EIOP.
6. EIOP gives the request to its disk driver software. This software issues the appropriate channel functions to the DCA channel adapter to send to the device.
7. Channel adapter translates to control signals for the disk. The disk writes data.
8. EIOP creates a response (d-packet) to the original D-packet and sends it to MUXIOP.
9. MUXIOP writes d-packet to mainframe. The interrupt caused by the packet read completion causes mainframe to see I/O completion.

IOS Model E: I/O Overview

Write Behind Example



"No" Write Behind Example



Y-MP EL differences summary

Not Updated 8.0

Most changes in kernel code are indicated by `#ifdef` preprocessor statements for `CRAYXMS` or `CRAYEL`, or for `MFSUBTYP` of `YMPEL`.

IOS

- EL systems can have from 1 to 16 IOS's (up to 4 per mainframe CPU, with IOS 0 being the master (MIOP)).
- EL IOS interface is very much like IOS Model B or C: one packet channel to each IOS. However, that channel is both a packet and data channel.
- All packet interfaces are patterned after IOS Model B/C/D except tape, which is patterned after Model E. All packets are 6 words.
- Packet types are mostly the same as Model B/C/D. Variants are:
  - D – UNIX-like tape driver
  - L – FDDI network driver
  - M – Ethernet driver
  - U – interface to a customer-written driver in the IOS (UNICOS device 45)
  - Y – provides address of variable-length E-packet for bmx tape
- EL IOS executes a third party real time operating system on its Heuricon MC68030-based processor.
- Operator interface is a terminal controlled by the IOS operating system.
- UNICOS is booted from an IOS disk. (Stand-alone boot kernel works, but is not released or supported.)
- The EL's IOS has an I/O Buffer Board (IOBB) strictly for IOS buffering. The IOBB is memory addressable from the IOP.
- Like the Model D, EL IOS can have a buffer memory for mainframe use. EL buffer memory consists of 1 or more (commonly 1–4) BMR boards and a controller board. Each BMR board is 16 megawords. This memory can be used as file system space or ldcache, just as buffer memory on an IOS Model D. Note one restriction: BMR cannot be used to ldcache a device residing on the same IOS.

Disk types

- Parameter file processing supports different disk types (DD3, DD4, DDAS2, RD-1 – see `md/pscan.c`).
- Disk slices are configured in blocks only, not cylinders.
- A-packets send a sector and track number of 1; real address is in the cylinder number field.

Tapes

- Mainframe communicates in fixed-length Y-packets, which provide the address of the variable length E-packets (patterned after Model E T-packets). The E-packets are moved as data.

CRAY Y-MP EL Hardware Summary

winchester SCSI disk
1/4" cartridge tape
Exabyte 8mm helical tape cassettes
Operator's Console

Not Updated 8.0

1–3 DD-3 ESDI disks — DC-3 controller
2–4 DR-1 removeable disks — DC-3 controller
1–4 IPI-2 disks — DC-4 controller
8 DD-3 disk array (+1 parity, +1 spare; hardware striped) — DAS-2 controller

IOS0
MC68030 IOP
IOBB (16MB)
VME bus
BMR (16MW x n)
BMR controller

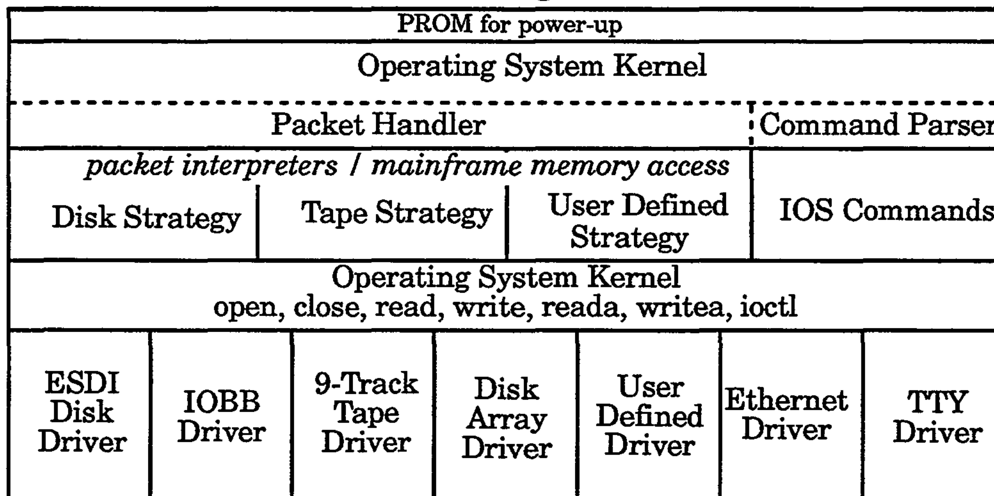
40MB/s (28MB/s sustained)
control & data channel

CRAY Y-MP EL
1–4 CPU's
32–128MW
30ns clock

TD-2 9-track tape — TC-2 controller
18-track cartridge (or silo) — SI-1 controller
Ethernet cable — EI-1 controller
FDDI (fiber optic) — FI-1 controller
HYPERchanel — HC-1 controller
HiPPI channel — HiPPI controller

IOS1
IOS2
(up to 4 IOS's per CPU)
IOS15

IOS Software Organization in an IOP



Not Updated 8.0

Networking hardware

- Model D hyperchannel driver was cloned to make Ethernet/FDDI driver (character device 44; new driver types are in `sys/netdev.h`).
- New DECnet driver (character device 46; DECnet protocol only through Ethernet currently).
- New NSC adapter type supported (N400 or A400; `sysnetdev.h`).
- A high speed HiPPI adapter will be supported later.

SSD

- Y-MP EL has no Solid-state Storage Device. Logical device caching can be done in buffer memory and central memory.

Mainframe hardware differences

- Bank conflicts during periods of heavy exchange activity caused memory problems; the pws[] was moved up to start exchange packages at 0nn20 boundaries so all exchange packages don't start in the same bank.
- The EL exchange package (XP) has a little different format.
- EL channels are numbered 0–71 (octal 107), but every other group of 8 is not hardware-useable.
- There is a slight difference in syndrome bit analysis in memory error correction.
- Only low 7 bits of the CI register are meaningful (others must be masked off).
- Master clear sets an XP flag that must be cleared at deadstart.
- At deadstart, monitor mode wait for channel completion during I/J packet handshaking must be longer.
- No hardware performance monitors exist on the EL. (HPM is defined as 0 in `sys/machd.ymp.h` for EL).

Mainframe software-only differences

- For the Y-MP EL there is a traditional UNIX tape driver (besides the UNICOS bmx and bmxdemon drivers). It is character device 43. This driver allows raw access to the tapes. That is, no label processing is done for the user.
- EL IOS can raise an MCU interrupt (for example, the hardware is there). But its software doesn't use it.

Binary release only

- Current (12/92) ELS release is number 2.2. It is based on UNICOS 6.1.
- Only `conf.SN.c`, `lowmem.c` and `config.h` files are released as source.
- Machine serial number is compiled into `conf.SN.c` and addressed as an `extern` from the rest of the kernel.
- All kernels are built for 128 MW memories; actual memory size is specified in the startup parameter file.

Y-MP system control

This module reviews the basic architecture of the CRAY Y-MP mainframe and relates these hardware features to the UNICOS system. Detail on differences for the CRAY C90 series, CRAY X-MP systems, and CRAY EL series follow. The following material references the foldout diagram "Y-MP System Control" at the end of this section. See the *CRAY Y-MP System Programmer Reference Manual*, publication CSM-0400-0A0, for additional information.

The diagram of the CRAY Y-MP mainframe is divided into three major sections as follows:

CPUn

- Each mainframe cpu (0–n) has this unique set.
- Shaded area highlights Exchange Package Information (described in a separate section).

Other shared resources

- Accessible by any CPU in system.
- Number of channels and clusters based on specific machine configuration.

Memory

- Common resource shared by all CPUs.
- Diagram shows major register save areas within kernel and user memory.

YMP – Foldout 2-61 } Chap does YMP & then we get differences on rest of pages.
p. 2-20 – 31

C-90 Differences Foldout 2-62
p. 32-34

X-MP Differences foldout 2-63
p. 35

Registers in each CPU

In the diagram the number under each box represents register size (in bits).

- **Program registers**
  - **A0–A7** Address registers

    Eight 32-bit register used primarily for address reference. Used by applications and kernel extensively.
  - **B00–B7** Intermediate address registers

    Sixty-four 32-bit registers for processing address register data. Used by applications and kernel extensively.
  - **S0–S7** Scalar registers

    Eight 64-bit scalar registers for integer and floating arithmetic. Used by applications and kernel extensively.
  - **T00–T77** Intermediate scalar registers

    Sixty-four 64-bit scalar vector registers for processing scalar data. Used by applications and kernel extensively.
  - **V0–V7** Vector registers

    Eight vectors each with sixty-four 64-bit floating point elements. Used by applications extensively but sparingly by the UNICOS kernel.
  - **VM** Vector mask

    Use by vector merge and test instructions to allow operations to be performed on individual vector elements.
  - **V** Vector length.

    Specifies the length of vector (number of elements) for all vector operations. Used by applications but not generally by the kernel.
  - **VNU** Vector not used bit

    Cleared when vectors are changed during a user interval. If it is still 1 when the kernel is entered, the kernel should not have to resave the user vectors. UNICOS kernel does not use VNU, because the kernel does not save all user vectors on every entry from a user process. Therefore at the time the kernel disconnects a process, VNU does not indicate whether vectors have not been used since the last full save, or merely during the last user interval.

- **Hardware performance monitor registers**
  - **HPM** Hardware performance monitor

    There are 4 sets of these registers, only 1 of which is "selected" for update by the hardware. (See the *CRAY Y-MP System Programmer Reference Manual*, publication CSM-0400-0A0 for more information). This is necessary to accurately account for wait-semaphore time. These counters advance only when the CPU is not in monitor mode and can be selected (/cleared) and read only in monitor mode. By default group 1 (hold issue conditions) is selected. See the HPM(4D) man page in *UNICOS File Formats and Special Files Reference Manual*, publication SR-2014, for information about the user interface to these registers.
- **Programmable clock**
  - **II** Interrupt Interval register

    Loaded with a given number of hardware clock periods by means of a PCI instruction. When the ICD counts down to 0 this II value is reloaded by the hardware into the ICD. This technique is used by the clocking CPU and user CPU profiling.
  - **ICD** Interrupt count down register

    Loaded at the same time as the II register via the PCI instruction. ICD is decremented by the hardware each clock period, and when zero, sets the flags register's PCI bit (this does not cause a CPU exchange in monitor mode).

    In general one CPU in the system has its programmable clock enabled to force at least one CPU to interrupt into the kernel each 1/60 second (minor clock cycle) for process scheduling. Any CPU can have its programmable clock set for specific times event processing such as user profiling and alarm signals.
- **Control registers**
  - **P** Program counter register

    Rightmost 2 bits address the 16 bit parcel of the word. On an exchange, P contains IBA-relative parcel address of the next instruction. On a return jump (R) instruction this value is stored in this CPU's B00 register. (Compilers restrict P to 2G parcels.)
  - **IBA/ILA** Instruction base address / Instruction limit address

    These registers can only be loaded via an exchange. The kernel sets them to the bounds of a process' s instruction (or "text") memory. P + IBA = absolute parcel address of the instruction to decode. A jump instruction loading P with a value such that P + IBA >= ILA results in setting the Program Range Error (PRE) flag, and an exchange.

    Six trailing zeroes are assumed by hardware, forcing instruction areas to begin on 0100 word boundaries. UNICOS allocates user instruction memory on an exact multiple of this value.

    CAUTION: the size of IBA/ILA determines the number of bits compared; any "garbage" to the left of that size is not considered during the compare with ILA; for example, use of a trashed B-register.
  - **DBA/DLA** Data base address / Data limit address

    These registers can only be loaded via an exchange. The kernel sets them to the bounds of a process' s data memory. Load or store instruction operands +

DBA = absolute word address of data to load or store.

A load or store operation in which the sum of the instruction operands + DBA >= DLA results in setting the Operand Range Error (ORE) flag, and an exchange (unless mode bit IOR is 0).

Six trailing zeroes are assumed by hardware, forcing data areas to begin on 0100 word boundaries. UNICOS allocates user memory on an exact multiple of this value

CAUTION: The size of DBA/DLA determines the number of bits compared; any "garbage" to the left of that size is not considered during the compare to DLA; for example, use of a trashed A-register. ♦

- **WS** Waiting on semaphore bit.

  WS indicates the CPU was holding issue on a test-and-set (SMjk 1,TS) instruction at the time of the exchange. WS is not used by UNICOS outside of displaying it for a read from `/proc`.

- **Mode register**

  Mode bits are mostly used by user processes. A user can select them through an `ioctl`(2) request to `/dev/cpu` (see *UNICOS File Formats and Special Files Reference Manual*, publication SR-2014, for more information).

  All but PS, FPS and SEI are selectable (but MM is available only to superuser). Many can be read as a "status register" (see below).

  - **AVL** Additional vector logical (also **ESVL**: Enable Second Vector Logical). (Not every system has a second vector logical unit.)

    If set, hardware chooses the second vector logical functional unit first. This speeds logical processing but may slow down code heavy in floating point multiplies due to sharing of logic paths with the floating point multiply unit.

  - **PS** Program state

    Not used by UNICOS (except to display it in a read from a /proc file). PS means nothing to the hardware.

  - **FP** Floating point error status

    Indicates to the user (via read of the status register) that a floating point error has occurred (regardless of IFP). It's reset by hardware only when floating point interrupts are enabled or disabled (EFI/DFI instructions). The kernel could reset it via the XP, but does not. FPS is not used by the kernel.

  - **BDM** Bi-directional memory

    If set, multiple word read and write operations (V, T, B block load/stores) can occur concurrently. This can result in indeterminate results when read/write areas overlap (for example, a load may begin before a store completes). Can be toggled by hardware instructions (EBM/DBM) and by `ioctl`(2) calls.

  - **IOR** Interrupt-on operand range error

    If set, load/store instructions where memory reference+ DBA >= DLA set the ORE flag and cause an exchange. IOR can be toggled by hardware instructions (ERI/DRI) and by `ioctl`(2). With ORE interrupts disabled, operand range errors result in a read of 0 or no-op write.

- **IFP** Interrupt-on floating point error

  If set, floating point errors set the FPE flag and cause an exchange. IFP can be toggled by hardware instructions (EFI/DFI) and by `ioctl`(2) calls. With floating point interrupts disabled, errors can be detected by reading the status register.

- **IUM** Interrupt-on uncorrectable memory error

  If set, multiple-bit memory errors fill in the Memory error information register and set the MEI flag, causing an interrupt. IUM must be set to get RPE interrupts.

- **ICM** Interrupt-on correctable memory error

  If set, single-bit memory errors fill in the Memory error information registers and set the MEI flag, causing an interrupt. Set for user processes except for special handling during memory error flooding. Not set for kernel except during memory error correcting.

Note: Single-bit error reads from memory yield correct results in the target register regardless of the ICM bit. Disabling MEI interrupt does not disable secded hardware.♦

- **EAM** Enhanced addressing mode

  If set, CPU addresses in native Y-MP 32-bit mode, otherwise CPU uses 24-bit X-MP compatibility mode. (See the section "Memory addressing modes" in this chapter for details.)

- **SEI** CPU selected for external interrupts

  If set and given all other CPU states equal, an I/O channel interrupt is directed to this CPU. Used only on C90 and a Y-MP with asymmetric CPU's. See the IOI flag description below for details on CPU selection by I/O channel interrupts.

- **IMM** Interrupt in monitor mode

  If IMM and MM are set, all interrupt types are enabled except IPI, PCI, MCU and IOI. UNICOS kernel runs in this mode (but with ICM disabled). In normal operation all kernel interrupts result in exchange with pw_xpux usually causing a panic.

- **MM** Monitor mode

  If set, the CPU is not interruptible by the 4 interrupt types mentioned above (see IMM), and can execute privileged instructions (load the XA, function channels, clear I/O interrupts, load the real time clock, send an interprocessor interrupt, etc.). When not in MM mode, the Y-MP CPU will no-op such instructions. UNICOS allows a superuser to set MM via `ioctl`(2) to `/dev/cpu`.

- **Flag bits**

  Flag bits are set by the hardware when an interrupting event occurs, unless the interrupt type has been disabled via a mode bit (above).

  When a flag bit is set, the CPU exchanges its active exchange package (XP) with the contents of the memory resident XP image pointed to by XA.

  The kernel must clear every flag before exchanging it back into the hardware registers, otherwise the same interrupt reoccurs. A more detailed description of the interrupt and what the UNICOS kernel does in reaction to each of these flags is detailed in Chapter 4, "Kernel Mainline".

  No exchange occurs if a CPU is in pure monitor mode (MM=1 but IMM=0); no flag is set.

  In interruptible monitor mode (IMM), for example, when executing the UNICOS kernel, the setting of any flag except IPI, PCI, MCU of IOI causes an exchange (with the XP image referenced by XA).

  – **RPE** Register parity error

    This flag is set when a parity error is detected in V, B, T, ST and SB registers and instruction buffers. RPECHIP status register is also filled in. IUM mode bit must be set to get RPE interrupts.

  – **IPI** Interprocessor interrupt (also known as ICP "Internal central processor"

    This flag is raised by execution of a MM-privileged SIPI instruction. CPUs in monitor mode are not interruptible for IPIs, though the IPI flag is not set immediately, the interrupt remains pending until the CPU goes to user mode.

  – **DL** DeadLock interrupt

    Deadlock is recognized by the hardware when all CPUs with the same cluster number are holding issue on test-and-set (SMjk 1,TS) instructions. They do not have to be testing the same semaphore. All such CPU's receive a DLI. This can happen frequently when a multitasking group of processes is not fully connected.

  – **PCI** Programmable clock interrupt

    Set when a CPU's ICD register counts down to 0. PCI will not interrupt monitor mode, but remains pending until the CPU returns to user mode.

  – **MCU** Maintenance control unit

    MCU is set by a hardware signal from the IOS. IOS software can send such a signal whenever it has a timeout on a central memory transfer and wants the mainframe to reduce any memory contention that may be causing the timeout. Y-MP hardware fans out the MCU interrupt to all CPUs. MCU interrupt will not interrupt monitor mode, but the interrupt remains pending until the CPU returns to user mode.

  – **FPE** Floating point error

    Any floating point instruction raises this flag when it detects an overflow or underflow condition (unless the IFP mode bit is 0).

  – **ORE** Operand range error

    Some data load/store instruction's memory address + DBA >= DLA. This interrupt is disabled if IOR mode bit is 0.

CAUTION: The size of DBA/DLA determines the number of bits compared; any "garbage" to the left of that size (from a trashed A register, for example) is not considered during the compare with DLA.

- **PRE** Program range error

  Some jump instruction's memory address + IBA >= ILA.

  This interrupt is disabled if IFP mode bit is 0.

  CAUTION: The size of IBA/ILA determines the number of bits compared; any "garbage" to the left of that size (from a trashed B register, for example) is not considered during the compare with ILA.♦

- **MEI** Memory error interrupt.

  All reads of data memory or an instruction buffer or a read of memory by an I/O channel (LOSP,HISP,VHISP) are monitored by single-error correction/ double-error detection (SECDED) hardware. Each 64-bit word has an 8-bit check byte stored with it.

  If a single-bit error is detected, it is corrected for delivery to the register or channel. If the ICM mode bit is on, the MEI bit is set, the Memory Error information registers are filled and an exchange occurs. If a double-bit (or detectable multiple bit) error is detected, no correction is done. If the IUM mode bit is on, the MEI bit is set, the memory error information registers are filled and an exchange occurs.

  In either case, memory error information is sent through the error channel to the error logger or MWS (during the next exchange sequence).

  Memory error interrupts can occur in pure monitor mode (if ICM/IUM enabled).

- **IOI** I/O Interrupt

  When a LOSP (6MB) or VHISP (1000MB) channel transfer completes it directs an I/O interrupt to the CPU of its choice.

  That choice is made according to the following priority:

  1. A CPU in monitor mode. The interrupt remains pending until cleared by that CPU or it enters user mode. A CPU in monitor mode can detect pending I/O interrupts by reading the CA register.
  2. A CPU with the SEI mode bit set (this mode bit is not used in UNICOS.
  3. A CPU holding issue on a Test and Set instruction.
  4. The CPU which last issued a clear interrupt(CI,Aj) for that channel.

- **EEI** Error Exit Interrupt

  Execution of the `00` opcode (either deliberately through the ERR assembler mnemonic, or accidentally through a jump into data). The kernel CAL panic macro uses this instruction to force an interrupt to the panic routine.

- **NEI** Normal Exit Interrupt

  Execution of the `EX` (`004`) opcode. In user code, this is a system call. The kernel uses the `EX` instruction in a few instances to do very special processing. It is also used by he kernel C Language panic macro uses this instruction to force an interrupt to the panic routine.

- **Status register**

  Instruction `073i01` (`Si SR0`) transmits the "status register" to the left-most bits of register `Si`.

  The status register is an aggregate of several mode register bits (PS FPS, IFP, IOR, BDM) plus 5 other fields:

  - **CL** Clustered

    Set to indicate that the CPU's cluster number is nonzero and therefore that the process may load/store from/to cluster registers.

    If 0, cluster number is zero and cluster reads deliver 0 and cluster writes no-op. The CL bit is not stored in the memory-resident exchange package.

  - **UME** Uncorrectable memory error

    Set when the error occurs, but is reset on an exchange or whenever the `SR0` is read (any `073`).

  - **CME** Correctable memory error

    This bit is set when the error occurs, but is reset on an exchange or whenever the `SR0` is read (any 073).

  - **PN** Processor number

    Always 0 if the CPU is not in monitor mode. Since all CPUs are equal, it should not be important to users which CPU is currently executing their code. A UNICOS process such as a a diagnostic is able to make an `ioctl`(2) call on `/dev/cpu` to request to be scheduled only on a particular CPU(s).

  - **CLN** Cluster Number

    `Si SR0` always delivers 0 for cluster number if the CPU is not in monitor mode.

    Since all clusters are equal and assigned to processes by the kernel, it should not be important to users which cluster they are addressing. It should only be necessary to know that a cluster is assigned, which can be known through the CL bit.

    A UNICOS process such as a diagnostic is able to make an `ioctl`(2) call on `/dev/cpu` to restrict itself to a particular cluster number(s).

- **Memory error information**

  These registers are filled in by the hardware on an MEI interrupt.

  - **E** Read Error Type

    These 2 bits flag the error as correctable / uncorrectable.

  - **S** Syndrome

    These 8 bits are the result of the exclusive-OR of the word's generated check byte and the original check byte.

  - **R** Read Mode

    On an X-MP, these 2 bits indicate type of memory reference (I/O, Scalar, Vector/B/T, instruction fetch/exchange). On a Y-MP, they must be used in conjunction with Port.

  - **P** Port (Y-MP only)

These 3 bits represent memory port in use (A, B, D), and combined with Read Mode indicate what type of operation was being performed. There are 10 valid combinations (see the *CRAY Y-MP System Programmer Reference Manual*, publication CSM-0400-0A0, for more information).

- **CS/B** Chip Select / Bank

  These bits (number varies by machine type) identify the memory chip and bank where the error was detected. Exact word address is not given. To locate and rewrite the word, the kernel must scan the chip for the error.

- **RPECHIP**

  These 6 bits are filled in by hardware on an RPE interrupt. They contain chip function and chip number.

- **Exchange address and exchange package information**

  XA Exchange address register

  Contains memory address of the XP image to be exchanged by the hardware with the active XP registers in the event of an interrupt.

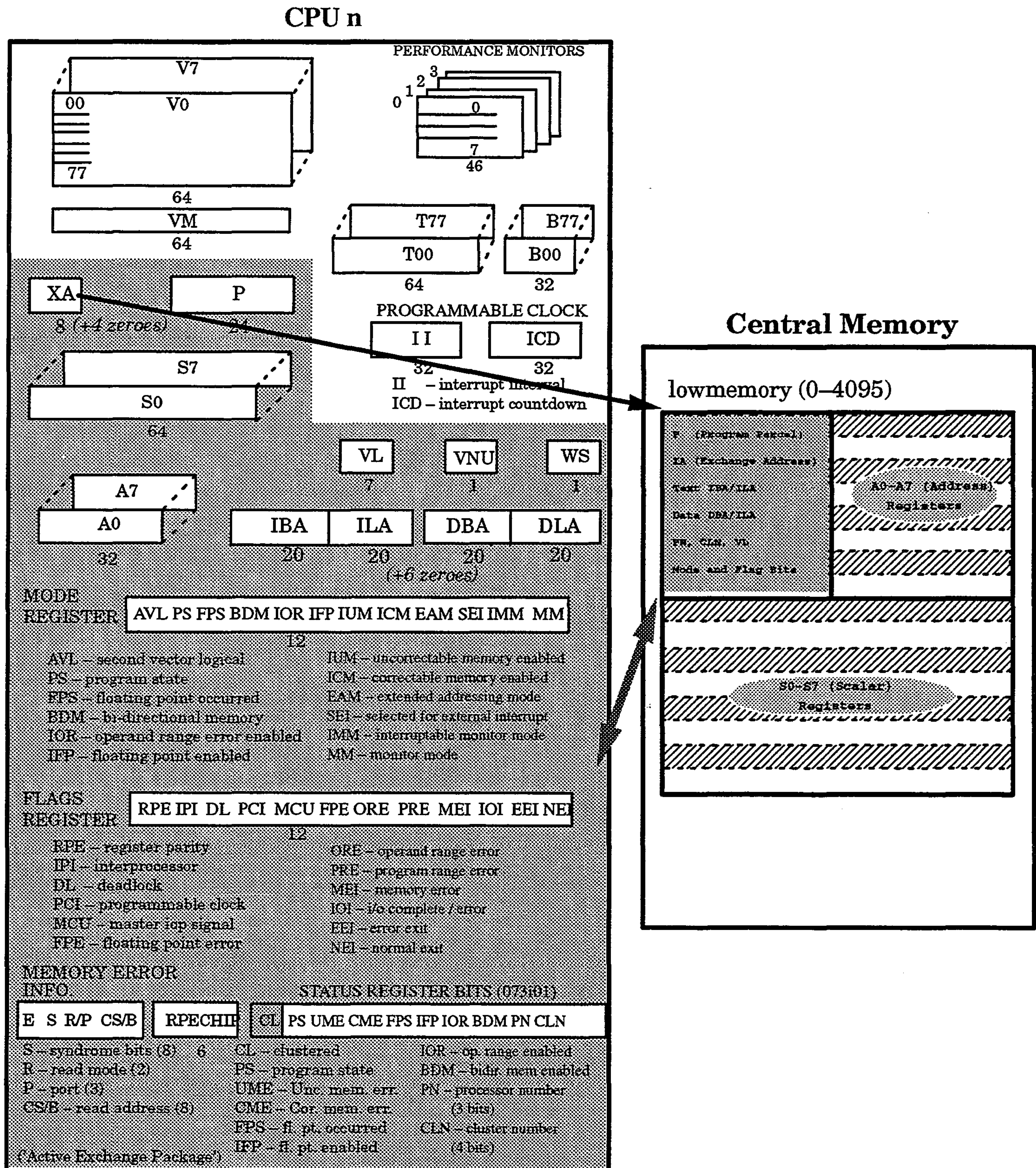
  Four zeros are appended to the 8 bit value aligning the area on an octal multiple of 20. The effective 12 bit size of the XA restricts the address to be within the first 4096 words of memory.

  XA can only be changed when in monitor mode.

  The figure on the right illustrates the exchange process.

  - The XA points to the area in memory used by the exchange.
  - Memory-resident XP images are 020 words in length
  - The contents of the CPU's registers shown in the shaded area are "packaged up" and stored in the XP area while simultaneously the previous contents of the XP area is loaded into the corresponding CPU registers.
  - The flags register in the XP in memory reflects the reason for the interrupt. Memory error information is stored for memory error interrupts.
  - The CPU (immediately) begins executing under the environment of the new register values. Note significantly the following:
    - ▲ The CPU switches to any new modes.
    - ▲ The base and limit addresses change to bound the new program area.
    - ▲ The new p address value provides the "next" instruction (based on IBA) to execute. (The exchange is effectively a branch).
  - In UNICOS, XA points at the CPU's own *cpu* entry in the PWS table – usually `pw_xpus` while in user mode, or `pw_xpux` while in system mode.

Exchange Illustration



Shared resources

- **Real time clock register**

  RTC Real time clock register

  RTC increments each hardware clock period. It can be set only in monitor mode. UNICOS kernel sets it to 0 at startup (it will not wrap for 14000 years with a 6ns clock). A system time is maintained in a memory address, in seconds since 1990.

- **Channels and channel registers**

  CI Channel Interrupt register

  Instruction `Ai CI` returns the number of the lowest numbered channel with a pending I/O interrupt. This number and pending interrupt are cleared with the hardware-privileged `CI,Ai` instruction. A 0 is returned if there are no pending interrupts.

  LOSP channels 6MB channel pairs

  Used for packets to/from the IOS. Even numbers are input, odd numbers are output. 8 pairs, 020–037 (Y-MP) Loading CA or CL registers are privileged operations.

  - **CL** Channel limit register

    Loaded with buffer end address +1.

  - **C** Channel address register

    When loaded with a buffer starting address the channel begins operation.

  - **C** Channel Error

    Parity error flag for input channel, unexpected Resume signal for output channels. Not checked on the mainframe end by UNICOS.

  HISP channels (not shown in diagram)

  Up to four 100/200MB channels used for data to/from an IOS model D. There is one HISP per IOC (IO cluster) on a Model E IOS. Under complete control of the IOS; no interrupts to the mainframe CPUs.

  VHISP channels 1000MB channels used for data to/from SSD(s)

  Bi-directional channels

  Channels 1 and 5 (+ 11 and 15 with second SSD) (Y-MP)

  Loading a CA or CL registers is a privileged operation.

  - **CA** Channel Address register

    First loaded with the SSD memory address (64-word block address)

    Second loaded with central memory buffer starting address

  - **BL** Block Length register

    Loaded with direction (high order bits) and number of 64 word SSD blocks. When loaded using CL command transfer begins.

  - **CE** Channel Error

    Parity error flag for input channel, unexpected Resume signal for output channels. Readable at any time. See the *CRAY Y-MP System Programmer Reference Manual*, publication CSM-0400-0A0, for more information.

- **Cluster registers**

  Cluster registers can pass data among CPUs quickly. They are used to coordinate kernel **multithreaded operation** and user **multitasking processing**.

  Systems are shipped with NCPU+1 (number of CPUs plus one) clusters. A CPU cluster number CLN of zero references no cluster. Cluster number 1 is reserved for the UNICOS kernel. Clusters 2–n are assigned to user processes (even if not multitasked).

  Each cluster consists of a set of 32 semaphore, 8 shared T, and 8 shared B registers as follows:

  - **SM0-37** Semaphore registers

    These 32 bits can be separately set/reset by any CPU "assigned to the cluster" (for example, with CLN = this cluster's number).

    They provide a hardware interlock via the "SMjk 1,TS" instruction where only 1 CPU can set the semaphore, other CPUs assigned to the same cluster testing the same semaphore, will hold issue on the test-and-set.

    When all CPUs in the same cluster are holding on a test-and-set (any semaphore) each CPU in the cluster receives a Deadlock interrupt (DLI) causing an exchange.

  - **ST0-7** Shared T registers

    Used to pass 64-bit scalar values from CPU to CPU quickly.

  - **SB0-7** Shared B registers

    Used to pass 32-bit integer (address) values from CPU to CPU quickly.

Y-MP C90 system control

The following information references the foldout diagram "Y-MP C90 System Control" diagram at the end of this section.

Significant differences with "Y-MP System Control" are noted below by each major section.

Several registers are larger (in bits) that in the Y-MP. The sizes are shown on the diagram but not generally noted in the detailed discussion.

Registers in each CPU

- **Program registers**
  - **V0-V7** Vector registers

    Each of the eight vector registers contains 128 elements.
  - **VM1** Vector mask

    A second vector mask is provided to cover the additional 64 vector elements.
  - **BMM** Bit matrix multiply

    Bit matrix multiply unit can be loaded via a vector register.
  - **Hardware performance monitor registers.**

    HPM Hardware performance monitor counters are extended to 48 bits (from 46). There is only one 32 element group. All are running at the same time.
- **Status registers** New class of registers
  - **VNU** Vector not used

    Bit is now reloaded from the memory copy of the exchange package. (Still not used since nearly every user exchange involves vector register usage).
  - **FPS** Floating point status

    Shows floating point error. Was a Mode register on Y-MP.
  - **WS** Waiting on semaphore

    Included in Status registers. Still only referenced by `/proc`.
  - **PS** Program status

    Was a Mode register on Y-MP.
- **Mode registers**

  There are two sets of mode bits on the C90. The interrupt mode bits of the Y-MP have been expanded and placed in a separate category.
  - **C90** Y-MPC90 mode must be set to recognize new C90 instructions (native mode).

    If 0, executes in Y-MP instructions (compatibility mode).
  - **ESL** Enable second vector logical

    Was AVL mode register on Y-MP.
  - **BDM** Bidirectional memory

    Same as Y-MP.

- **MM** Monitor mode

  Same as Y-MP.

- **Interrupt mode and interrupt flag registers**

  Each mode corresponds to an interrupt type and interrupt flag (16 of them).

  On a Y-MP only interrupts IOR, IFP, IUM, and ICM were "maskable". In addition a CPU in Monitor Mode (MM) was interruptible only if IMM was set (MM=1 and IMM=1)

  On a C90 the function of the IMM bit has been replaced by the CPU Enable Interrupt Mode **EIM** flag (not shown).

  - The EIM is set on exchanges to non-monitor mode (user) and cleared on exchanges to monitor mode (kernel).
  - While in MM the instruction EMI sets the EIM bit (enabled) and DMI clears the bit (disable). Interrupt Modes FNX, FEX, and IPR are not affected by this bit (they are always enabled).
  - Interrupts PCI, ICP, RTI, MCU, MEC, BPI, ORE, FPE, MEU, and RPE are held pending if EIM if clear.
  - If EIM is set, interrupts, or held interrupts corresponding to **set** interrupt modes, are allowed; held interrupts,.except PCI and ICP, are cleared on any exchange.
  - The UNICOS kernel executes the EMI command immediately upon entry to the kernel (exchange user–>kernel). Mode Registers IOR, IFP, IUM, IRP, IPR, FNX, and FEX are set on the exchange-in providing for the same interrupt processing in the kernel as for a Y-MP. (See "Kernel Mainline – immtrap" for detail on kernel interrupt processing.)

    The following registers are changed or new on the C90:

    - ▲ **IBP/BPI** (Enable) breakpoint interrupt

      A write reference to an address within the breakpoint range (BP instruction).

    - ▲ **IRT/RTI** (Enable) real time interrupt

      The hardware for this is not (yet) available. UNICOS runs with this disabled.

    - ▲ **IIP/ICP** (Enable) internal CPU interrupt

      Renamed IPI interrupt of Y-MP.

    - ▲ **IMI/MII** (Enable) monitor mode interrupt

      Interrupt cause by attempt to execute a privileged instruction in non-monitor mode (user). The Y-MP treated such cases as a no-op. For the C90 UNICOS runs with this mode disabled.

    - ▲ **IDL/DLI** (Enable) deadlock interrupt

      ON a C90 a CPU entering monitor mode remembers its user mode cluster. Other CPU's holding issue on test-and-set instructions in that cluster will not get a deadlock interrupt.

    - ▲ **IUM/MEU** and **ICM/MEC** (Enable) uncorrectable and correctable memory errors interrupts

Are separate flags on C90, was only MEI "Memory Error" on Y-MP. Memory correction is Single-byte correction/double-byte detection (SBCDBD). A "byte" is 4 bits. A hardware word is 80 bits (not 72), giving the 16 check bits.

- **SEI** (Enable) System I/O Interrupts (not shown)

  I/O interrupts on all CPU are disabled after the first until re-enabled by setting this flag with the ESI command. (See Section 4, Kernel Mainline – "`usrioi`" the I/O interrupt handler, for more information.)

Status registers

The single status register on the Y-MP is replaced by 8 registers on the C90.

- **SR0** General – refer to the "C90 System Programmer Reference Manual" for detail.
- **SR1** Undefined.
- **SR2** HPM counters 00–17.
- **SR3** HPM counters 20–37.
- **SR4** Memory error type data.
- **SR5** Memory error syndrome data.
- **SR6** Memory error address data.
- **SR7** RPE status data.

Shared resources

- **Cluster registers**

  Extended to 17 clusters for 16 CPUs.

- Channels and channel registers

  Both low-speed and VHISP channels are numbered differently than on a Y-MP.

  The CE (Channel error, or Channel status words) are all enlarged to 32 bits.

X-MP system control

The following information references the "X-MP System Control" foldout.

Significant differences to "Y-MP System Control" are noted below by each major section.

Several registers are smaller (in bits) that in the CRAY Y-MP. The sizes are shown on the diagram but not generally noted in the detailed discussion.

Registers in each CPU

- Mode Registers

  AVL, PS, and EAM are "Modes" on the CRAY X-MP. WS is a mode.

  These registers are different than equivalent registers on a CRAY Y-MP:

  - **EMA** Extended memory addressing

    When set provides full X-MP 24 bit addressing (native mode). When not set provides 22 bit addressing (compatibility mode).

  - **AVL** Additional vector logical

    Not a "Mode Register" but just a separate CPU status flag.

  - **WS** Waiting on semaphore

    A Mode register on the CRAY X-MP.

- **Flags register registers**

  Interrupt and interrupt flag RPE is not present on the X-MP.

- **Memory error information**

  RPECHIP is not present on the CRAY X-MP.

- **Status registers**

  Elements UME and CME are not in the status register.

Shared resources

- **Cluster registers**

  3 clusters for 1 and 2 CPU system, 5 clusters for 4 CPU systems..

- **Channels and channel registers**

  Both low-speed and VHISP channels are numbered differently than on a CRAY Y-MP.

Interrupt processing summary

The figure on the right shows major hardware register components in relation to the memory areas they are saved to and restored from during the interrupt handling process.

The "cycle" is started with the CPU connected to a user process (executing the user program in non-monitor mode MM=0). The CPU's XA would be pointing to this CPU's PWS cpu entry field `pw_xpus` (xp user). Refer to the letter key for the following.

- **Exchange in**

  x When any interrupt occurs the CPU performs an exchange sequence, saving the exchange package data at the XA address `pw_xpus` while simultaneously loading the XP data into the CPU. Note that the A and S registers are saved by the action.

  The XA is then set to point to this CPU's `pw_xpux` (UNIX exchange package).

- **Kernel entry**

  b The B registers are saved in the user field `u_saveb`.

  t The T registers are saved in the user field `u_savet`.

  v If the interrupt is not a normal exchange (NEX – or system call) the VMs and V0 (only) are saved in the user `u_savevm`, `u_savevml`, and `u_savev` fields.

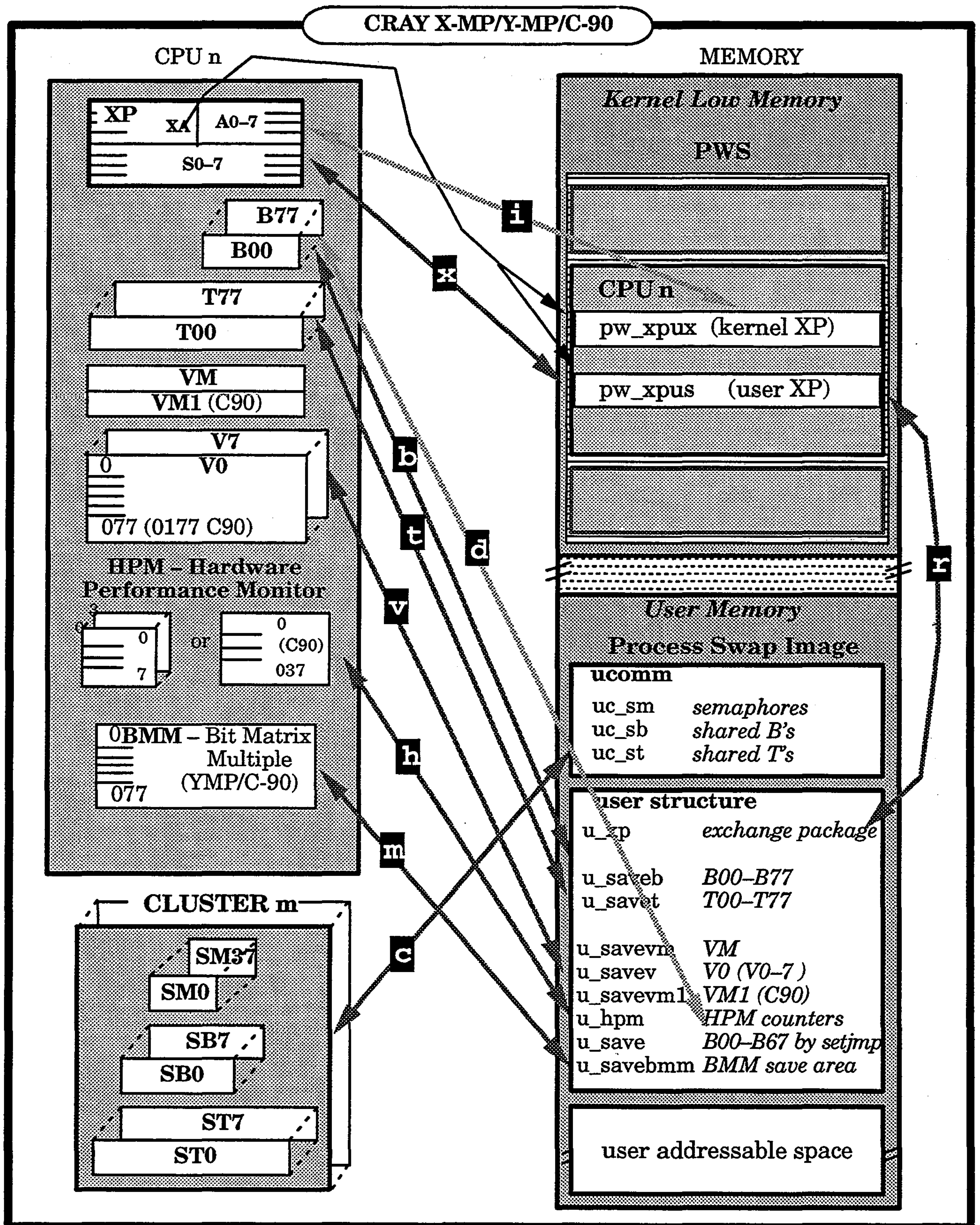
  Vectors V1–7 are saved by the kernel for non-NEX interrupts only if used by the kernel routine.

- **Kernel exchange**

  While the CPU is executing kernel code the XA is pointing to this CPU's `pw_xpux` area. A CPU in the kernel is operating with IMM on (X-MP/Y-MP) or EIM set (C90) so exchanges CAN take place.

  i If an interrupt occurs the CPU performs an exchange sequence, saving the exchange package data at the XA address `pw_xpux` while simultaneously loading the XP data into the CPU.

  The `pw_xpux` p address is set to the address of `immtrap`. Executing `immtrap` normally results in a system panic – all interrupts enabled in kernel (monitor) mode are catastrophic events.



CRAY X-MP/Y-MP/C-90
CPU n
MEMORY
XP
XA
A0–7
S0–7
B77
B00
T77
T00
VM
VM1 (C90)
V7
0
V0
077 (0177 C90)
HPM – Hardware Performance Monitor
0
7
or
0
(C90)
037
0BMM – Bit Matrix Multiple (YMP/C-90)
077
CLUSTER m
SM37
SM0
SB7
SB0
ST7
ST0
Kernel Low Memory
PWS
CPU n
pw_xpux (kernel XP)
pw_xpus (user XP)
User Memory
Process Swap Image
ucomm
uc_sm semaphores
uc_sb shared B's
uc_st shared T's
user structure
u_xp exchange package
u_saveb B00–B77
u_savet T00–T77
u_savevm VM
u_savev V0 (V0–7)
u_savevm1 VM1 (C90)
u_hpm HPM counters
u_save B00–B67 by setjmp
u_savebmm BMM save area
user addressable space
i
x
b
t
d
v
h
m
c
r

- **Process switch**

  During the processing of an interrupt the kernel may decide to disconnect the current process and resume (connect) to another process. A "context switch" takes place.

  For the process being disconnected the following information is saved (see the figure on the right):

  v Any vectors not yet saved are saved in u_savev.

  h The HPM data is saved in u_hpm.

  m BMM data is saved in u_savebmm.

  c If this is the last member of a multitasked group save the cluster information in ucomm uc_sm, uc_sb, and uc_st fields.

  d The kernel's active B and T registers are saved in users u_save.

  Select the resumed process's user/ucomm area – resume this process:

  r The XP in pw_xpus is saved in the old user's u_xp field. Restore the XP data in u_xp for the resumed process.

  h The HPM data is loaded from u_hpm.

  c If this is the first member of a multitasked group select a cluster and load the cluster information from uc_sm, uc_sb, and uc_st fields.

  d The kernel's saved B and T registers are reloaded from u_save.

- **Kernel exit**

  b The B registers are loaded from user field u_saveb.

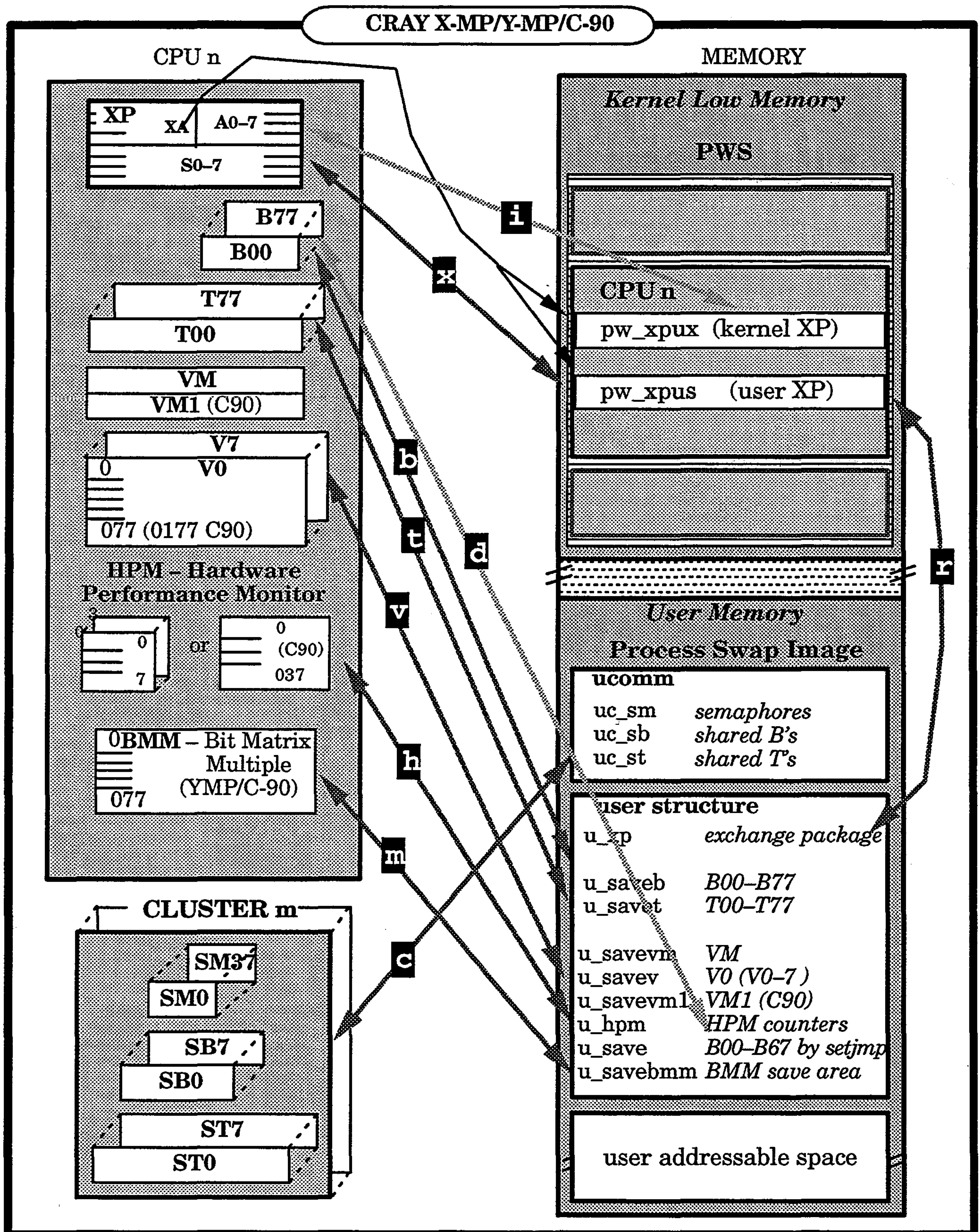
  t The T registers are loaded from user field u_savet.

  v If the interrupt is not a normal exchange (NEX – or system call) VMs and V0–7 are loaded from u_savevm, u_savevm1, and u_savev fields. If NEX set VM, V0–7 registers are cleared to zeros

- **Exchange out**

  Just before returning to the user, the XA is loaded with the address of this CPU's pw_xpus, now containing the interrupted or newly resumed process's exchange package information.

  x The kernel executes an EX instruction causing an NEX interrupt. The CPU exchange package (user) data is loaded into the CPU while the kernel's XP data is saved in pw_xpus. The A and S registers are restored for the user here.


CRAY X-MP/Y-MP/C-90
CPU n
MEMORY
XP
XA
A0–7
S0–7
B77
B00
T77
T00
VM
VM1 (C90)
V7
V0
0
077 (0177 C90)
HPM – Hardware Performance Monitor
0
7
or
0
(C90)
037
0BMM – Bit Matrix Multiple (YMP/C-90)
077
CLUSTER m
SM37
SM0
SB7
SB0
ST7
ST0
Kernel Low Memory
PWS
CPU n
pw_xpux (kernel XP)
pw_xpus (user XP)
User Memory
Process Swap Image
ucomm
uc_sm semaphores
uc_sb shared B's
uc_st shared T's
user structure
u_xp exchange package
u_saveb B00–B77
u_savet T00–T77
u_savevm VM
u_savev V0 (V0–7)
u_savevm1 VM1 (C90)
u_hpm HPM counters
u_save B00–B67 by setjmp
u_savebmm BMM save area
user addressable space
i
x
b
t
d
v
h
m
c
r

Exchange (XP information)

Exchange package crash display

The facing page shows sample exchange package contents using the `crash`(8) `xp` display output.

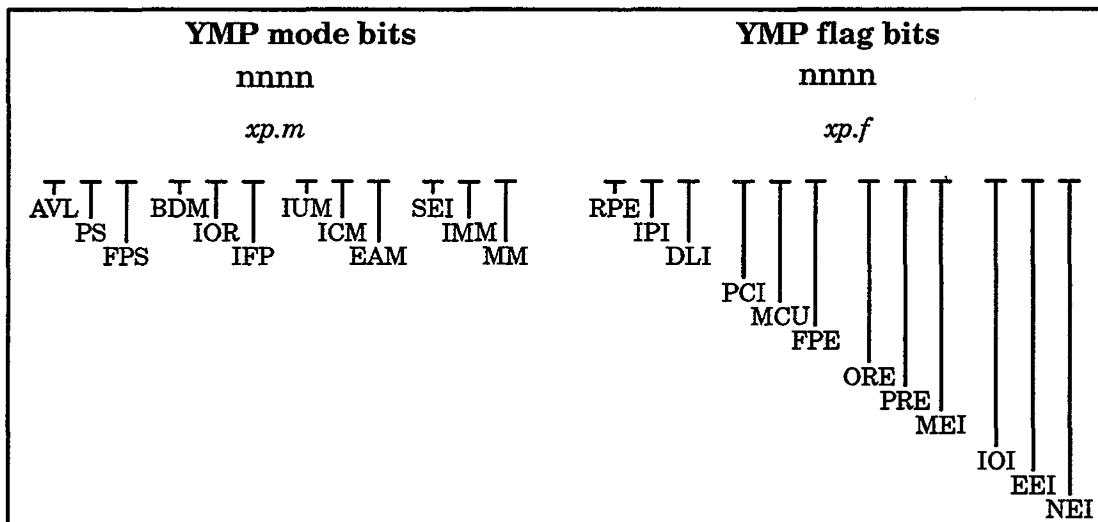
Sample Cray Y-MP C90 and Cray Y-MP displays from `crash`(8) release 7.0 show the full breakdown of all XP fields including each "Interrupt" Mode, Flag, Status, and "Modes" bits.

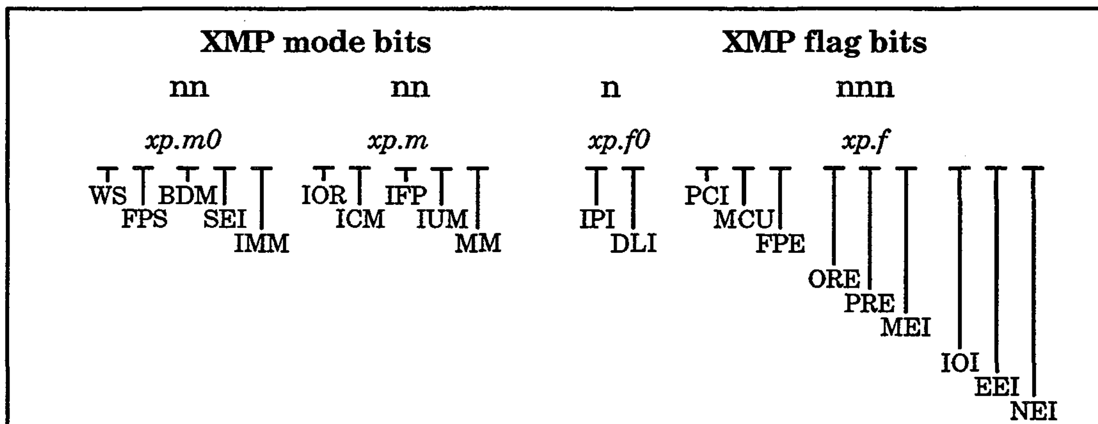
The third CRAY X-MP sample is from a release 5.0 crash. This version older version shows the mode "m" and flags "f" as 2 octal fields each, leaving the user to interpret the bits individually.

A breakdown of the mode and flags bits for the CRAY Y-MP and CRAY X-MP follows.

Refer to the "System Control" information earlier on in this chapter for the meaning of the abbreviations.

XP mode and status bit breakdown





CRAY Y-MP C90 XP

```
Exchange package at address 03036002 for cpu 0 - Cray C90

  p           0a  a0 00000000000              Interrupt
  iba   03056000  a1 00000000000          Modes      Flags      Status
  ila   03076000  a2 00000000000          00410      00000       00
  xa        0660  a3 00000000000          irp=0      rpe=0      vnu=0
  dba   03056000  a4 00000000000          ium=0      meu=0      fps=0
  dla   03076000  a5 00000000000          ifp=0      fpe=0      ws =0
  vl           0  a6 00000000000          ior=0      ore=0      ps =0
                  a7 00000000000          ipr=0      pre=0
                                          fex=0      eex=0
  s0  0000000000000000000000  ........    ibp=0      bpi=0      Modes
  s1  0000000000000000000023  ........    icm=1      mec=0       00
  s2  0000000000000000000000  ........   imc=1      mcu=0      c90=0
  s3  0000000000000000000000  ........    irt=0      rti=0      esl=0
  s4  0000000000000000000000  ........    iip=0      icp=0      bdm=0
  s5  0000000000000000000000  ........    iio=0      ioi=0      mm =0
  s6  0000000000000000000000  ........    ipc=1      pci=0
  s7  0000000000000000000000  ........    idl=0      dl =0
                                          imi=0      mii=0
  cluster 11    vnu 0                     fnx=0      nex=0
```

CRAY Y-MP XP

```
Exchange package at address 03473002 for cpu 0 - Cray Y/MP

  p           0a  a0 00000000000          Modes      Flags
  iba   03513000  a1 00000000000           0020      0 0000
  ila   03533000  a2 00000000000          esl=0      ws =0
  xa        0660  a3 00000000000          ps =0
  dba   03513000  a4 00000000000          fps=0      icp=0
  dla   03533000  a5 00000000000          bdm=0      dl =0
  vl           0  a6 00000000000          ior=0      pci=0
                  a7 00000000000          ifp=0      mcu=0
                                          ium=0      fpe=0
  s0  0000000000000000000000  ........    icm=1      ore=0
  s1  0000000000000000000013  ........    eam=0      pre=0
  s2  0000000000000000000000  ........    sei=0      me =0
  s3  0000000000000000000000  ........    imm=0      ioi=0
  s4  0000000000000000000000  ........    mm =0      eex=0
  s5  0000000000000000000000  ........               nex=0
  s6  0000000000000000000000  ........
  s7  0000000000000000000000  ........

  cluster 5    vnu 0

  Error type = None      Syndrome = 0
  Error csy = 0          Port = 0
  Error csb = 0          Read mode = 0
```

CRAY X-MP XP

```
p        30233c cpu  2        a0 00047450
ib      3110400  il  3177000  a1 00041617
m       4  36  vnu 0          a2 00047445
xa          660  vl    3      a3 00000014
f       0    0 ps 0  cln 2    a4 00041612
db      3110400  dl  3177000  a5 00047454
e  0     rm  0  syn       0   a6 00047446
------------- and so on ---------------
```

Memory addressing modes

CRAY-1 and CRAY X-MP without EMA

All CRAY-1s and those CRAY X-MPs with up to 4MW of memory (through serial 122, 217 and 308) had no need for EMA hardware.

Their maximum memory size of 017777777 words could be addressed by these 22-bit addresses:

- Word address in the P register,
- IBA/ILA and DBA/DLA,
- Word address in all jump, load, and store instructions.
- All negative values could be held as simple 2's complements of 4MW.

CRAY-1 binaries may not be executed on the:

X-MP in compatibility mode:	UNICOS will not allow a CRAY-1 binary to be executed on anything but a CRAY-1.
X-MP in EMA mode:	The EMA hardware would treat any expressions over 2MW in the load/store instructions as negative.
Y-MP:	In compatibility mode – the EAM hardware treats any expressions over 2MW in the load/store instructions as negative, as above. In native EAM mode – the EAM hardware misinterprets load/store instructions as 3-parcels.

A CRAY-1 may not execute a binary targeted for the:

X-MP in EMA mode or Y-MP in compatibility mode	EMA-compatible code produces wild jumps. The EMA 24-bit A-load instruction looks like a conditional jump instruction to a CRAY-1. (See EMA Instruction Formats, below).
Y-MP in EAM mode:	Also yields unpredictable results. The EAM 3-parcel register load and store instructions are misinterpreted as 2-parcel load and store instructions. (See EAM Instruction Formats, below).

Maximum address is `017777777` (4MW)

- **Jumps**

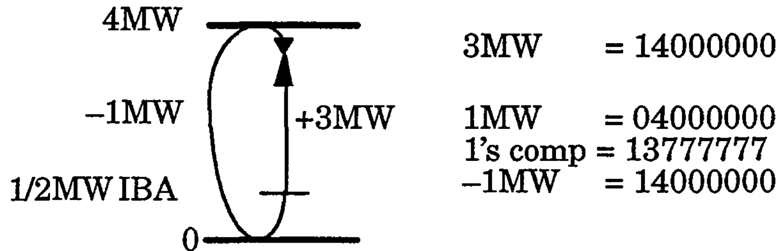
  Jumps contain 24-bit parcel address expressions (or use 24 bits of a B register). The 22-bit word address portion limits the value to 4MW.

  CAL flags any expression over 4MW or under –2MW as truncated; negative expressions are simply 4MW complements.

  For example: –1MW = 3MW, –2MW = 2MW –4MW = 0, and.–5MW = –1MW. A CAL expression of 5MW would be truncated to 1MW.

- **IBA**

  The IBA is a 22-bit word address. The effective address addition (exp + IBA) is done in 22-bit hardware, limiting the result to 4MW.



  Note that the effect of a 3MW jump or a –1MW jump are the same, for example, their values as 22 bits are identical.

- **Register loads and stores**

  Loads and stores contain 22-bit word addresses, and use just 22 bits of register Ah.

  The Ah register can address anywhere in 4MW and the immediate expression can do the same.

- **DBA**

The DBA is a 22-bit word address.

The effective address addition (exp + Ah + DBA) is done in 22-bit hardware, limiting the result to 4MW in the same way as for jumps (above).

- **Address loads**

  Address-loading instructions contain 22-bit word address expressions.

A pre-EMA machine will not recognize the new 24-bit address-loading instruction (`01i`) but will execute it as a conditional jump (the Ai register signifying the condition) with unpredictable results.

CRAY X-MP in compatibility modes

CRAY X-MP CPU with EMA hardware is in Cray-1 "compatibility mode" when their exchange package EMA bit is 0.

CRAY X-MPs starting with serial numbers 123, 218, and 309 have EMA hardware.

Their maximum memory size of 16MW (`077777777`) words cannot be addressed by the 22-bit word address in the P register, the 22-bit word address in all jump instructions, or the 22-bit word address in non-EMA load and store instructions.

Advantages of running in compatibility mode:

The only advantage of this mode is to allow CAL programs written with load/store expressions; > 2MW, to be recompiled without change and run on an EMA machine.

Disadvantages of running in compatibility mode:

- The program can only address 4MW of memory
- UNICOS must keep the program from spanning any 4MW memory boundary.

CRAY X-MP compatibility mode binaries may not be executed on the:

CRAY-1:	Though physically okay, UNICOS does not permit it.
X-MP in EMA mode:	It's load/store address expressions over 2MW would be interpreted as negative
Y-MP in either mode:	If an EAM machine is in compatibility mode it would act just like an EMA machine described above. If an EAM machine is in EAM mode it would misinterpret the binary's load/store instructions.

An CRAY X-MP in compatibility mode may not execute a binary targeted for the:

X-MP in EMA mode:	Hardware recognizes the binary's EMA 24-bit A-load instruction, but all memory references "wrap" at 4MW boundaries.
Y-MP in EAM mode:	Also yields unpredictable results. The EAM 3-parcel register load and store instructions are misinterpreted as 2-parcel load and store instructions. (See EAM Instruction Formats, below).

An X-MP is in Cray-1 "compatibility mode" if the exchange package EMA bit is 0.

With EMA, hardware supports memory sizes up to 16MW (`077777777` – the maximum value of 24 bits).

- **Jumps**

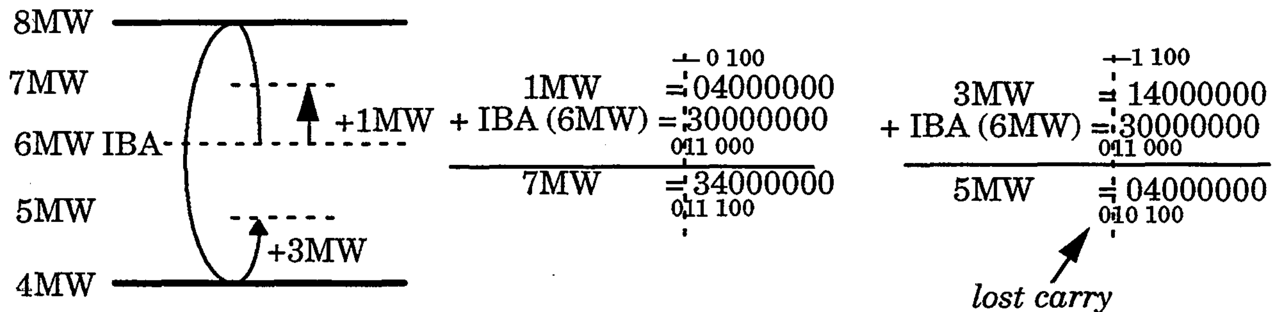
  Jumps contain 24-bit parcel address expressions, limiting code size to 4MW as in pre-EMA machines.

- **IBA**

  The IBA is a 24-bit word address, capable of addressing to 16MW.

  The effective address addition (exp + IBA) is done in 22-bit hardware, truncating any carry.

  The left 2 bits of the IBA don't participate in the add, but are copied into the 24-bit result. The net effect of this is illustrated below:



Note that all jumps behave as if the program is running in a 4MW machine; the program cannot address across a 4MW boundary. (UNICOS is responsible for guaranteeing that any task with an exchange package in compatibility mode does not cross a 4MW boundary.)

- **Register loads and stores**

  The hardware treats the 22-bit expressions and 22 bits of register Ah the same as for jumps. Register Ah and the expression can range from +4MW to –4MW.

- **DBA**

  The DBA is a 24-bit word address but only 22 bits participate in the add. The result is the same as for jump instructions; the program data must be within a 4MW section of memory, not crossing a 4MW boundary.

- **Address loads**

  Either 22 or 24-bit expressions can be loaded into Ah – it makes no difference, as only 22 bits are used by the hardware.

  Hardware can tell the difference between the new 01i 24-bit A-load and the `0110–017` jump instructions, so either A-load is okay.

An X-MP is in 24-bit "extended" mode if the exchange package EMA bit is 1. In extended mode a program can address the full 16MW potential memory size.

- **Jumps**

  Jumps contain 24-bit parcel address expressions, limiting program size to 4MW as in every combination of hardware and mode.

- **IBA**

The IBA is a 24-bit word address, capable of addressing to 16MW.

The effective address addition (exp + IBA) is done in 24-bit hardware. The 22 bits in the instruction are zero filled to 24 bits before the add to the IBA.

The result is illustrated below: (assume the IBA is 7MW)

Note that all jumps can cross 4MW boundaries but are limited to +4MW from the IBA. This limits all EMA mode executable codes to 4MW.

- **Register loads and stores**

  The hardware uses 24 bits of Ah, and sign extends the 22-bit expressions to 24 bits for the add. This is done to allow the expression to be negative.

  The Ah register can address any address in memory.

  The expression provides an offset of +2MW to –2MW. Any expression over 21 bits (2MW) is handled as if negative.

Note: The loaders prohibit any code with a relocatable expression over 2MW in a load or store from running in EMA mode. (Relocated expressions are always positive.) A programmer may code an absolute expression over 2MW in a load or store and the loaders will allow the code to run in EMA mode; it is the programmer's responsibility to realize that expressions over 2MW is interpreted by the EMA hardware as negative. ♦

- **DBA**

  The DBA is a 24-bit word address, all 24 bits of which participate in the add. The result is that exp + Ah + DBA can address 16MW words of memory.

- **Address loads**

  A 24-bit address is needed in the Ah to address all 16MW. Opcodes 020–040 create zero-filled 24-bit fields from the 22 bits in Ai or Si, so they are okay in extended mode for addressing up to 4MW.

  The 24-bit A-load instruction (opcode 01i) must be used for addressing beyond 4MW from the DBA.

CRAY X-MP EMA instruction formats

In the table are the bit formats of the instructions affected by EMA.

- **New instruction:**

  All but the last one (containing the 24-bit expression) are identical on a CRAY-1 or any CRAY X-MP.

  The 24-bit A-load instruction was added to EMA machines so that 16 MW addresses could be coded into instructions.

  An X-MP with EMA hardware recognizes the new instruction in either EMA or compatibility mode. However, a machine without EMA hardware will not test the 8th bit in such an instruction and thus misinterprets it as a jump.

- **Expansion of 22-bit expressions:**

  The 22-bit expressions in the jump instructions are zero-filled by the hardware before the add to the longer IBA in an EMA machine.

  But in order to preserve the ability to code negative offsets in load/store instructions, the 22-bit expressions in the load and stores are sign-filled before the add to the longer DBA in an EMA machine. Hence the caution about expressions longer than 21 bits.

Powers of 8	Significant Maximums	Megawords
1 = 1 10 = 8 100 = 64 1000 = 512 10000 = 4,096 100000 = 32,768 1000000 = 262,144 10000000 = 2,097,152 100000000 = 16,777,216	07777777 = 2,097,151 (21 bit EMA relocatable expression) 17777777 = 4,194,303 (22 bit word address) 77777777 = 16,777,215 (24 bit EMA word address)	04000000 = 1MW 10000000 = 2MW 14000000 = 3MW 20000000 = 4MW 24000000 = 5MW 30000000 = 6MW 34000000 = 7MW 40000000 = 8MW 44000000 = 9MW 50000000 = 10MW 54000000 = 11MW 60000000 = 12MW 64000000 = 13MW 70000000 = 14MW 74000000 = 15MW 100000000 = 16MW

Instruction Formats (relevant to EMA)

	g(4)	h(3)	i(3)	j(3)	k(3)	m(16)
J	00	6	0	22-bit word		2-bit parcel
R	00	7				
JAZ	01	0				
JAN	01	1				
JAP	01	2				
JAM	01	3				
JSZ	01	4				
JSN	01	5				
JSP	01	6				
JSM	01	7				
Ai exp,Ah	10	Ah	Ai/Si	22-bit word		
exp,Ah Ai	11					
Si exp,Ah	12					
exp,Ah Si	13					
Ai exp	02	0	Ai/Si	22-bit word		
Si exp	04					
Ai exp *(implemented via LONGALD opdef)*	01	Ai	1	24-bit word		

Testing the hardware for EMA

```
        S1           0
        S0           1
        VWD          7/O'015,1/1,D'24/P.NOEMA
        S1           -1
NOEMA    =           *
        W@MCEMA,A2   S1
*if EMA hardware present, S1 = -1, else S1 = 0
```

JSN NOEMA
–or–
*A5 NOEMA*4*

CRAY Y-MP compatibility (24-bit) mode

EAM is enhanced addressing mode. The Y-MP (or X-MP EA) has 32-bit address registers.

CRAY Y-MP CPUs are in X-MP "compatibility mode" when their exchange package EAM bit is 0. The CRAY Y-MP EAM mode is completely compatible with a CRAY X-MP in its 24-bit addressing EMA mode and may be referred to as either "X mode" or "24-bit mode".

Why use compatibility mode?

The only advantage of this mode is to enable a Y-MP system to execute a CRAY X-MP EMA system binary. (There is a very slight saving of memory space because of 2-parcel instead of 3-parcel load and store instructions).

Disadvantages:

24-bit addresses can only address 16MW of memory. **UNICOS must keep it from spanning any 16MW memory boundary.** The binary cannot address beyond its own 16MW partition of memory (see the description on the facing page).

CRAY Y-MP compatibility mode binaries may not be executed on the following:

CRAY-1:	Would interpret long A-load instructions as jumps
X-MP in compatibility mode:	Would recognize the long A-loads, but the binary's memory references would "wrap" at 4MW boundaries.
Y-MP in EAM mode:	Would misinterpret the binary's 2-parcel load/store instructions

A CRAY Y-MP in compatibility mode may not execute a binary for the following:

CRAY-1:	UNICOS does not allow it, and it may not work anyway. EAM hardware would treat expressions over 2MW in load/store instructions as negative
X-MP in compatibility mode:	Same reason as above.
Y-MP in EAM mode:	Would yield unpredictable results. EAM 3-parcel register load and store instructions would be interpreted as 2-parcel load and store instructions. (See EAM Instruction formats, below).

A CRAY Y-MP (or CRAY X-MP EA) CPU is in X-MP "compatibility mode" when its EAM bit is 0.

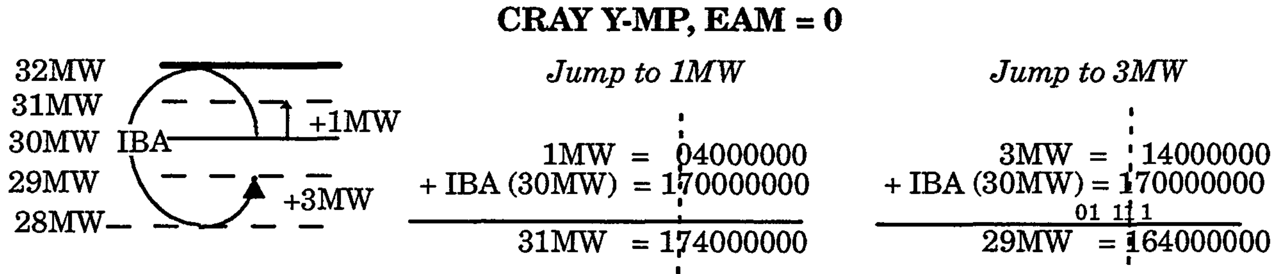
- **Jumps**

  Jump instructions contain 24-bit parcel address expressions, limiting code size to 4MW as in all combinations of hardware and mode. The P register is still 24 bits.

- **The IBA**

  The IBA is a 26-bit word address, capable of addressing to 64MW. The effective address addition (22-bit word expression + IBA) is done in 22-bit hardware. Any carries are truncated. The left 4 bits of the IBA don't participate in the add, but are copied into the 26-bit result.

  The net effect of this is shown below (assume the IBA is 30MW):



  All jumps behave as if the program is running in a 16MW machine. That is, the program cannot address across a 16MW boundary. An attempt to do so causes a wrap within a 4MW region. (UNICOS guarantees that a program in compatibility mode does not cross a 16MW boundary.)

- **Register loads and stores**

  Hardware executes a 24-bit expression load into an A register just like an X-MP in extended mode.

  Other loads and stores are also interpreted identical to an X-MP in extended mode. That is, sign extending the 22-bit expression to 24 bits for the add. This allows the expression to be negative.

  Register Ah plus the expression can range from +16MW to –16MW.

  The 22-bit expression, when sign filled to 26, provides an offset of –2MW to +2MW. Any expression over 21 bits (2MW) is effectively negative.

Note: The loaders prohibits any code with a relocatable expression over 2MW in a load or store from running in "X" mode. (Relocated expressions are always positive.) A programmer may code an absolute expression over 2MW in a load or store and the loaders will still allow the code to run in "X" mode; it is the programmer's responsibility to realize that any expression over 2MW is interpreted by the hardware as negative. ♦

- **DBA**

  The DBA is a 26-bit word address but only 24 bits participate in the add. The left 2 bits are merely copied into the result.

  The effect is that the program data must be within a 16MW section of memory, not crossing a 16MW boundary. UNICOS ensures this for an "X" mode program.

- **Address Loads**

  Registers can be loaded with addresses up to 16MW in the same way as on an XMP in extended mode. The EA hardware recognizes the long (24-bit) A register load instruction.

CRAY Y-MP EAM (32-bit) mode

CRAY Y-MP CPUs are in "EAM mode" when their exchange package EAM bit is 1.

In EAM mode, program code size is still limited to 4MW, but data size can range up to (theoretical) 4GW because of the 32-bit registers. Running in EMA mode is preferable for this reason.

This "native" Y-MP mode is also known as "Y", "EAM", or "32-bit" mode.

Disadvantages:

- Programs from a CRAY X-MP must be recompiled or modified/reassembled since hardware will not recognize the 2-parcel load/store instructions or the EMA 24-bit A-load instruction..
- Program binary sizes grow very slightly due to the 3-parcel instructions.

CRAY Y-MP EAM mode binaries may not be executed on the following:

CRAY-1 or CRAY X-MP:	No CRAY machine prior to a CRAY Y-MP (or CRAY Y-MP EA) recognizes the 3-parcel load/store instructions.

A CRAY Y-MP in EAM mode may not execute a binary for the following:

CRAY-1 or CRAY X-MP:	Its 2-parcel load/store instructions would be misinterpreted.
X-MP:	Same reason as above, plus the misinterpretation of any 24-bit A-load instructions

A CRAY Y-MP (or CRAY X-MP EA) is in "enhanced" mode if the exchange package bit EAM is 1.

- **Jumps**

  Jump instructions still contain 24-bit parcel address expressions, limiting program size to 4MW.

  The P register is still 24 bits.

- **IBA**

  The IBA is a 26 to 32-bit word address, capable of addressing a theoretical 4GW of memory. The effective address addition (exp + IBA) is done in 26 to 32-bit hardware. The 22 bits in the instruction are zero-filled to the size of the IBA before the add to the IBA.

  The result is illustrated below: (assume the IBA is 13MW):

CRAY Y-MP, EAM = 1

J 1MW	J 3MW	J 4MW-1
04000000 1MW	14000000 3MW	17777777 4MW-1
+ 064000000 13MW (IBA)	+ 064000000 13MW (IBA)	+ 064000000 13MW (IBA)
070000000 14MW	100000000 16MW	103777777 17MW-1

  All jumps can cross 16MW boundaries but are limited to +4MW from the IBA. This limits all executable codes to 4MW.

- **Register loads and stores**

  The hardware recognizes opcodes `020`, `021`, `040`, `041`, `10h`, `11h`, `12h` and `13h` (register loads and stores) to be new 3 parcel instructions containing 32-bit expressions.

- **DBA**

  The DBA is a 26 to 32-bit word address. All bits participate in the add. The result is that exp + Ah + DBA can address any word up to a 4GW theoretical maximum. Any upper bits of the 32-bit expression and Ah register which do not exist in the DBA are considered 0.

- **Address loads**

  A 26 to 32-bit address is needed in the Ah to address all of memory. Opcodes `020`–`040` will be interpreted as 3-parcel instructions, containing 32-bit expressions.

  The 24-bit A load instruction (opcode `01i`) which was new to the CRAY X-MP with EMA is of no use to the CRAY Y-MP in extended mode and is not allowed (result is undefined).

CRAY Y-MP EAM instruction formats

In the table are the bit formats of the instructions affected by EAM.

New Instructions:

For the CRAY Y-MP in native (EAM) mode the A and S register load/store instructions have all been enlarged to 3 parcels in order to contain 32-bit expressions.

If a cpu is running in CRAY X-MP compatible mode (EAM=0) however, these opcodes are interpreted identically to a CRAY X-MP with EMA.

The 24-bit A-load instruction is unneeded, and undefined, for a CRAY Y-MP in EAM mode.

Compatibility mode expansion of 22-bit expressions:

A CRAY Y-MP in compatibility mode behaves exactly like a CRAY X-MP in EMA mode. The 22-bit expressions in jump instructions are zero-filled by hardware for the add to the longer IBA.

The 22-bit expressions in load and stores are sign-filled for the add to the longer DBA. Thus the caution about expressions longer than 21 bits also applies to the CRAY Y-MP in compatibility mode.

Powers of 8

Octal	Decimal
1 =	1
10 =	8
100 =	64
1000 =	512
10000 =	4,096
100000 =	32,768
1000000 =	262,144
10000000 =	2,097,152
100000000 =	16,777,216
1000000000 =	134,217,728
10000000000 =	1,073,741,824
100000000000 =	8,589,934,592

Significant Maximums

07777777 = 2,097,151
(21 bit EMA relocatable expression)

17777777 = 4,194,303
(22 bit word address)

77777777 = 16,777,215
(24 bit word address)

377777777 = 67,108,863
(26 bit word address)

37777777777 = 4,294,967,295
(32 bit word address)

Megawords

Octal	Size	Octal	Size
04000000 =	1MW	74000000 =	15MW
10000000 =	2MW	100000000 =	16MW
14000000 =	3MW	120000000 =	20MW
20000000 =	4MW	170000000 =	30MW
24000000 =	5MW	240000000 =	40MW
30000000 =	6MW	310000000 =	50MW
34000000 =	7MW	360000000 =	60MW
40000000 =	8MW	400000000 =	64MW
44000000 =	9MW	1000000000 =	128MW
50000000 =	10MW	2000000000 =	256MW
54000000 =	11MW	4000000000 =	512MW
60000000 =	12MW	10000000000 =	1GW
64000000 =	13MW	20000000000 =	2GW
70000000 =	14MW	40000000000 =	4 GW

Instruction Formats (relevant to EAM modes)

Instruction	g(4)	h(3)	i(3)	j(3)	k(3)	m(16)
J	00	6	0	22-bit word		2-bit parcel
R	00	7	0	22-bit word		2-bit parcel
JAZ	01	0	0	22-bit word		2-bit parcel
JAN	01	1	0	22-bit word		2-bit parcel
JAP	01	2	0	22-bit word		2-bit parcel
JAM	01	3	0	22-bit word		2-bit parcel
JSZ	01	4	0	22-bit word		2-bit parcel
JSN	01	5	0	22-bit word		2-bit parcel
JSP	01	6	0	22-bit word		2-bit parcel
JSM	01	7	0	22-bit word		2-bit parcel

(X mode formats)

Instruction	g(4)	h(3)	i(3)	j(3)	k(3)	m(16)
Ai exp,Ah	10	Ah	Ai/Si	22-bit word		
exp,Ah Ai	11	Ah	Ai/Si	22-bit word		
Si exp,Ah	12	Ah	Ai/Si	22-bit word		
exp,Ah Si	13	Ah	Ai/Si	22-bit word		
Ai exp	02	0/1	Ai/Si	22-bit word		
Si exp	04	0/1	Ai/Si	22-bit word		
Ai exp	01	Ai	1	24-bit word		

(Y mode formats)

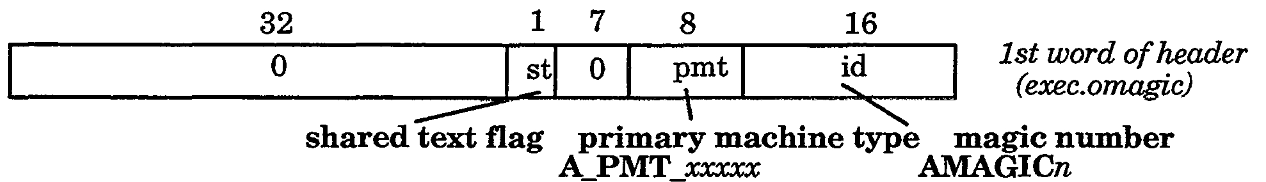
Instruction	g(4)	h(3)	i(3)	j(3)	k(3)	m(16)	n(16)
Ai exp,Ah	10	Ah	Ai/Si	0	0	32-bit word	
exp,Ah Ai	11	Ah	Ai/Si	0	0	32-bit word	
Si exp,Ah	12	Ah	Ai/Si	0	0	32-bit word	
exp,Ah Si	13	Ah	Ai/Si	0	0	32-bit word	
Ai exp	02	0/1	Ai/Si	0	0	32-bit word	
Si exp	04	0/1	Ai/Si	0	0	32-bit word	

Summary of hardware types / binary restrictions

Hardware mode flags in the CRAY X-MP and CRAY Y-MP exchange packages are similarly named, but are not identical. Summarized, they are as follows:

	CRAY-1 mode (22-bit)	**X-MP mode (24-bit)**	**Y-MP mode (32-bit)**
X-MP EMA	0	1	
Y-MP EAM		0	1

When binaries are linked, segldr flags their “primary machine type” in the file header:



A_PMT_xxxxx	pmt	Description
A_PMT_UNDF	0	Undefined (old form; magic number only)
A_PMT_INC	1	Incremental load code fragment
A_PMT_CRAY1	2	CRAY-1S
A_PMT_XMP_NOEMA	3	CRAY X-MP, 22-bit mode
A_PMT_XMP_ANY	4	CRAY X-MP, mode indifferent (no exp >21 bits)
A_PMT_XMP_EMA	5	CRAY X-MP, 24-bit mode
A_PMT_CRAY2	6	CRAY-2
A_PMT_YMP	7	CRAY Y-MP
A_PMT_C90	8	CRAY Y-MP C90

Prior to 6.0, binaries were classified by magic number (AMAGICn) only:

AMAGICn	ID	Description
A_MAGIC1	0407	CRAY X-MP, mixed text/data
A_MAGIC2	0410	CRAY X-MP shared text
A_MAGIC3	0411	CRAY Y-MP 32-bit, mixed text/data
A_MAGIC4	0412	CRAY Y-MP 32-bit, shared text

	X-MP mode	**Y-MP mode**
Text/Data	**407**	**411**
Shareable Text	**410**	**412**

On the facing page are summaries of which binary types can execute on which hardware and mode.

HARDWARE COMPATIBILITY

binary meant for:	executed on: CRAY-1 (or X-MP w/o EMA)	CRAY X-MP EMA=0	CRAY X-MP EMA=1	CRAY Y-MP EAM=0	CRAY Y-MP EAM=1
CRAY-1 (or X-MP w/o EMA)	**OK**	**OK**	expressions over 2MW treated negative	expressions over 2MW treated negative	load/store instructions misinterpreted
CRAY X-MP EMA=0	**OK**	**OK**	expressions over 2MW treated negative	expressions over 2MW treated negative	load/store instructions misinterpreted
CRAY X-MP EMA=1	Long A loads executed as jumps	Data references wrap at 4MW boundaries	**OK**	**OK**	load/store/long A load instructions misinterpreted
CRAY Y-MP EAM=0	Long A loads executed as jumps	Data references wrap at 4MW boundaries	**OK**	**OK**	load/store/long A load instructions misinterpreted
CRAY Y-MP EAM=1	3-parcel instructions misinterpreted	3-parcel instructions misinterpreted	3-parcel instructions misinterpreted	3-parcel instructions misinterpreted	**OK**

SOFTWARE RESTRICTIONS

binary flagged A_PMT_xxx	executed on: CRAY-1	X-MP (no EMA)	X-MP EMA=0	X-MP EMA=1	Y-MP EAM=0	Y-MP EAM=1
_CRAY-1	(407) **OK**	Not allowed	Not allowed	Not allowed	Not allowed	Not allowed
XMP NOEMA	Not allowed	**OK**	**OK** within 4MW partition	Not allowed	Not allowed	Not allowed
_XMP_ANY	Not allowed	**OK**	Not allowed	**OK**	**OK** within 16MW partition	Not allowed
_XMP_EMA	Not allowed	Not allowed	Not allowed	(407,410) **OK**	(407,410) **OK** within 16MW partition	Not allowed
_YMP	Not allowed	Not allowed	Not allowed	Not allowed	Not allowed	(411,412) **OK**

CRAY Y-MP C90 in native mode

The CRAY Y-MP C90 has new instructions such as EMI, ESI, breakpoint, VM1 register, semaphore test and branch, B read-and-increment, and 3-parcel jump instructions with 32-bit addresses (allowing code up to 1 gigaword).

CRAY Y-MP C90 in compatibility mode

If the CRAY Y-MP C90 mode bit in the exchange package is set, the C90 CPU is executing in Y-MP compatibility mode.

None of the new C90 instructions are recognized (note especially the 3-parcel jump instruction (with 32-bit address).

A 24-bit Y-MP jump address is zero-filled to 32 bits and the add to the IBA is done in 32-bit hardware. There are therefore no "boundaries", for example, no wrapping to the beginning of a partition. The CRAY Y-MP binary may be executed anywhere in memory.

But block memory transfers work a little differently on a CRAY Y-MP C90 in compatibility mode than on a CRAY Y-MP. Certain memory strides cause bi-directional memory results different than the compilers plan for on a CRAY Y-MP. Therefore, currently (8/92) Y-MP binaries must be run with BDM (bi-directional memory) mode off.

CRAY X-MP in compatibility mode

CRAY X-MP CPUs with EMA hardware are in "EMA mode" when their exchange package EMA bit is 1.

With Extended Memory Addressing hardware, program code size is still limited to 4MW, but data size can range up to 16MW. Running a X-MP in EMA mode is preferable for this reason.

This "native" X-MP mode behaves the same as the Y-MPs X-MP "compatibility" mode.

Disadvantage: The only restriction in this mode is that load/store 22-bit address expressions not exceed 2MW, because of the sign-extending done for data reference instructions.

CRAY X-MP EMA mode binaries may not be executed on the following:

CRAY-1:	It would misinterpret long A-load instructions as jumps.
X-MP in compatibility mode:	It would recognize the long A-load instructions, but the binary's memory references would "wrap" at 4MW boundaries.
Y-MP in EAM mode:	It would misinterpret the binary's 2-parcel load/ store instructions.

A CRAY X-MP in EMA mode may not execute a binary targeted for the following:

CRAY-1 or CRAY X-MP in compatibility mode:	Any expressions over 2MW would be misinterpreted as negative.
Y-MP in EAM mode:	Also yields unpredictable results. The EAM 3-parcel register load and store instructions would be misinterpreted as 2-parcel load and store instructions. (See EAM Instruction Formats, below).

Hardware system control foldouts

This section contains the following hardware system control foldout diagrams:

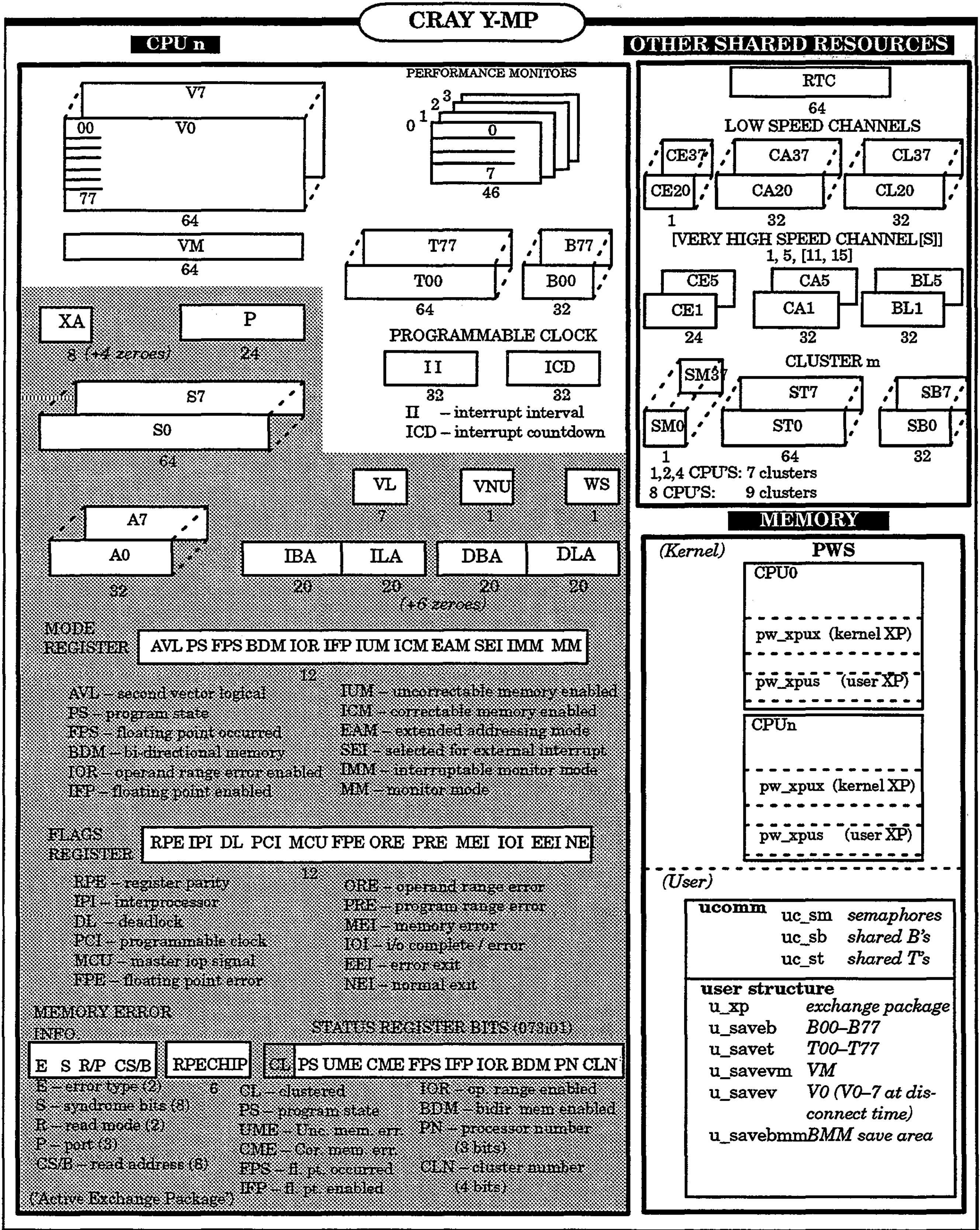
CRAY Y-MP System Control

CRAY Y-MP C90 System Control

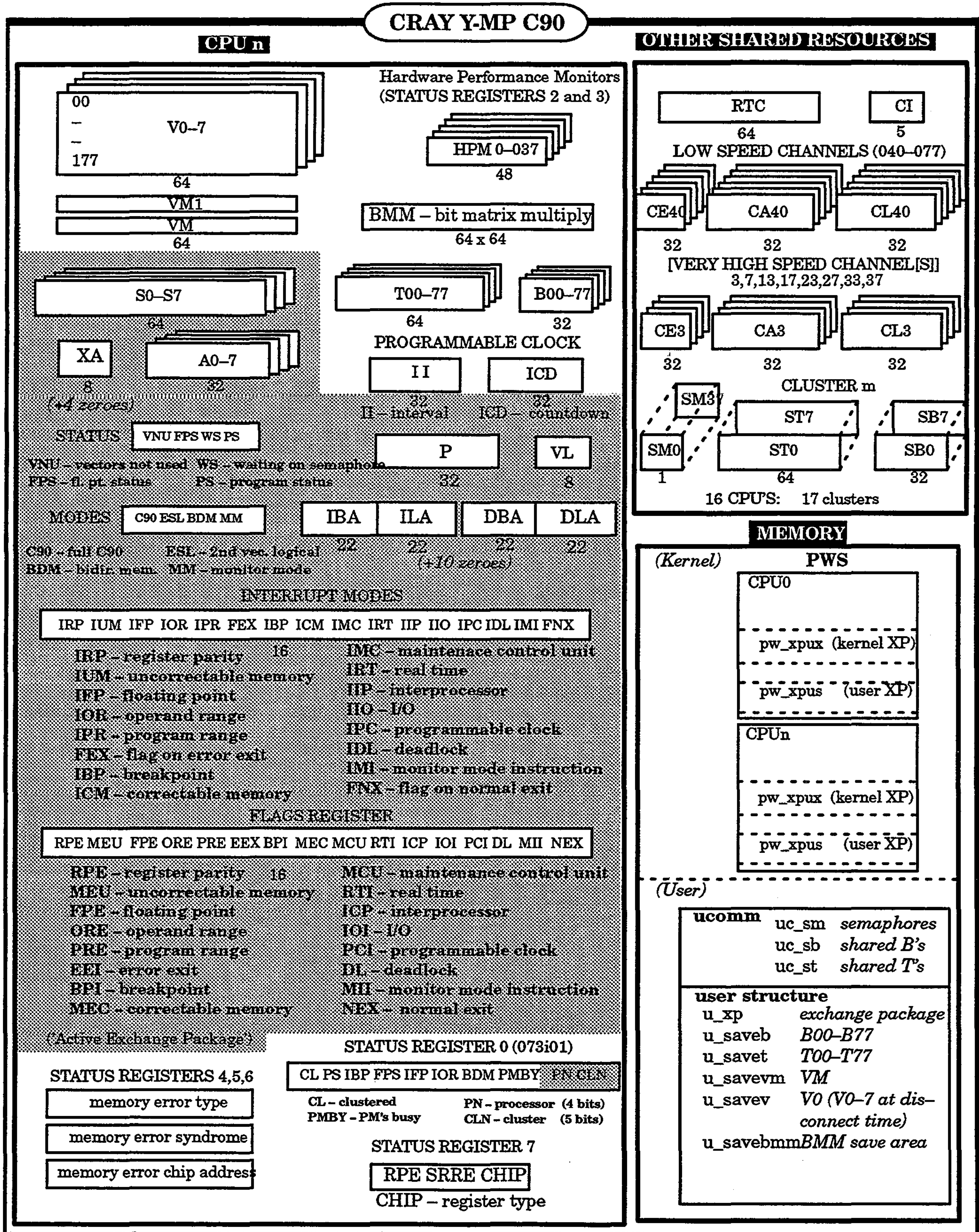
CRAY X-MP System Control

Interrupt Mode/Flag Summary (Y-MP C90)

This page used for alignment



CRAY Y-MP
CPU n
OTHER SHARED RESOURCES
PERFORMANCE MONITORS
V7
V0
00
77
64
VM
64
XA
P
8 (+4 zeroes)
24
S7
S0
64
T77
T00
64
B77
B00
32
PROGRAMMABLE CLOCK
II
ICD
32
32
II – interrupt interval
ICD – interrupt countdown
VL
VNU
WS
7
1
1
A7
A0
32
IBA
ILA
DBA
DLA
20
20
20
20
(+6 zeroes)
MODE REGISTER
AVL PS FPS BDM IOR IFP IUM ICM EAM SEI IMM MM
12
AVL – second vector logical
PS – program state
FPS – floating point occurred
BDM – bi-directional memory
IOR – operand range error enabled
IFP – floating point enabled
IUM – uncorrectable memory enabled
ICM – correctable memory enabled
EAM – extended addressing mode
SEI – selected for external interrupt
IMM – interruptable monitor mode
MM – monitor mode
FLAGS REGISTER
RPE IPI DL PCI MCU FPE ORE PRE MEI IOI EEI NEI
12
RPE – register parity
IPI – interprocessor
DL – deadlock
PCI – programmable clock
MCU – master iop signal
FPE – floating point error
ORE – operand range error
PRE – program range error
MEI – memory error
IOI – i/o complete / error
EEI – error exit
NEI – normal exit
MEMORY ERROR INFO.
STATUS REGISTER BITS (073i01)
E S R/P CS/B
RPECHIP
CL PS UME CME FPS IFP IOR BDM PN CLN
6
E – error type (2)
S – syndrome bits (8)
R – read mode (2)
P – port (3)
CS/B – read address (8)
CL – clustered
PS – program state
UME – Unc. mem. err.
CME – Cor. mem. err.
FPS – fl. pt. occurred
IFP – fl. pt. enabled
IOR – op. range enabled
BDM – bidir. mem enabled
PN – processor number (3 bits)
CLN – cluster number (4 bits)
("Active Exchange Package")
RTC
64
LOW SPEED CHANNELS
CE37
CE20
CA37
CA20
CL37
CL20
1
32
32
[VERY HIGH SPEED CHANNEL[S]]
1, 5, [11, 15]
CE5
CE1
CA5
CA1
BL5
BL1
24
32
32
CLUSTER m
SM37
SM0
ST7
ST0
SB7
SB0
1
64
32
1,2,4 CPU'S: 7 clusters
8 CPU'S: 9 clusters
MEMORY
(Kernel)
PWS
CPU0
pw_xpux (kernel XP)
pw_xpus (user XP)
CPUn
pw_xpux (kernel XP)
pw_xpus (user XP)
(User)
ucomm
uc_sm semaphores
uc_sb shared B's
uc_st shared T's
user structure
u_xp exchange package
u_saveb B00–B77
u_savet T00–T77
u_savevm VM
u_savev V0 (V0–7 at dis-connect time)
u_savebmm BMM save area


CRAY Y-MP C90
CPU n
00
V0–7
177
64
VM1
VM
64
Hardware Performance Monitors
(STATUS REGISTERS 2 and 3)
HPM 0–037
48
BMM – bit matrix multiply
64 x 64
S0–S7
64
XA
8
A0–7
32
(+4 zeroes)
T00–77
64
B00–77
32
PROGRAMMABLE CLOCK
I I
ICD
32
32
II – interval
ICD – countdown
STATUS
VNU FPS WS PS
VNU – vectors not used
WS – waiting on semaphore
FPS – fl. pt. status
PS – program status
P
32
VL
8
MODES
C90 ESL BDM MM
C90 – full C90
ESL – 2nd vec. logical
BDM – bidir. mem.
MM – monitor mode
IBA
ILA
DBA
DLA
22
22
22
22
(+10 zeroes)
INTERRUPT MODES
IRP IUM IFP IOR IPR FEX IBP ICM IMC IRT IIP IIO IPC IDL IMI FNX
16
IRP – register parity
IUM – uncorrectable memory
IFP – floating point
IOR – operand range
IPR – program range
FEX – flag on error exit
IBP – breakpoint
ICM – correctable memory
IMC – maintenace control unit
IRT – real time
IIP – interprocessor
IIO – I/O
IPC – programmable clock
IDL – deadlock
IMI – monitor mode instruction
FNX – flag on normal exit
FLAGS REGISTER
RPE MEU FPE ORE PRE EEX BPI MEC MCU RTI ICP IOI PCI DL MII NEX
16
RPE – register parity
MEU – uncorrectable memory
FPE – floating point
ORE – operand range
PRE – program range
EEI – error exit
BPI – breakpoint
MEC – correctable memory
MCU – maintenance control unit
RTI – real time
ICP – interprocessor
IOI – I/O
PCI – programmable clock
DL – deadlock
MII – monitor mode instruction
NEX – normal exit
('Active Exchange Package')
STATUS REGISTERS 4,5,6
memory error type
memory error syndrome
memory error chip address
STATUS REGISTER 0 (073i01)
CL PS IBP FPS IFP IOR BDM PMBY PN CLN
CL – clustered
PMBY – PM's busy
PN – processor (4 bits)
CLN – cluster (5 bits)
STATUS REGISTER 7
RPE SRRE CHIP
CHIP – register type
OTHER SHARED RESOURCES
RTC
64
CI
5
LOW SPEED CHANNELS (040–077)
CE40
CA40
CL40
32
32
32
[VERY HIGH SPEED CHANNEL[S]]
3,7,13,17,23,27,33,37
CE3
CA3
CL3
32
32
32
CLUSTER m
SM37
SM0
1
ST7
ST0
64
SB7
SB0
32
16 CPU'S: 17 clusters
MEMORY
(Kernel)
PWS
CPU0
pw_xpux (kernel XP)
pw_xpus (user XP)
CPUn
pw_xpux (kernel XP)
pw_xpus (user XP)
(User)
ucomm
uc_sm semaphores
uc_sb shared B's
uc_st shared T's
user structure
u_xp exchange package
u_saveb B00–B77
u_savet T00–T77
u_savevm VM
u_savev V0 (V0–7 at dis–connect time)
u_savebmm BMM save area

CRAY X-MP

CPU 1

PERFORMANCE MONITORS

V7
V0
00
77
64
0 1 2 3
0
7
46

VM
64

T77 B77
T00 B00
64 24

VL 7 VNU 1 AVL 1

PROGRAMMABLE CLOCK

II 32 ICD 32

II – interrupt interval
ICD – interrupt countdown

S7
S0
64

A7
A0
24

P 24 XA 8 EMA 1

IBA 17,18 ILA 17,18 DBA 17,18 DLA 17,18

MODE REGISTER: WS FPS BDM SEI IMM IOR ICM IFP IUM MM
10

- WS – waiting for semaphore
- FPS – floating point occurred
- BDM – bi-directional memory
- SEI – selected for external interrupts
- IMM – interrupt monitor mode
- IOR – operand range enabled
- ICM – correctable memory enabled
- IFP – floating point enabled
- IUM – uncorrectable memory enabled
- MM – monitor mode

FLAGS REGISTER: ICP DL PCI MCU FPE ORE PRE ME IOI EEX NEX
11

- ICP – interprocessor
- DL – deadlock
- PCI – programmable clock
- MCU – master iop signal
- FPE – floating point error
- ORE – operand range error
- PRE – program range error
- ME – memory error
- IOI – i/o complete / error
- EEX – error exit
- NEX – normal exit

MEMORY ERROR INFO.

E S R CS/B

- E – error type (2)
- S – syndrome bits (11)
- R – read mode (2)
- CS/B – read address (10)

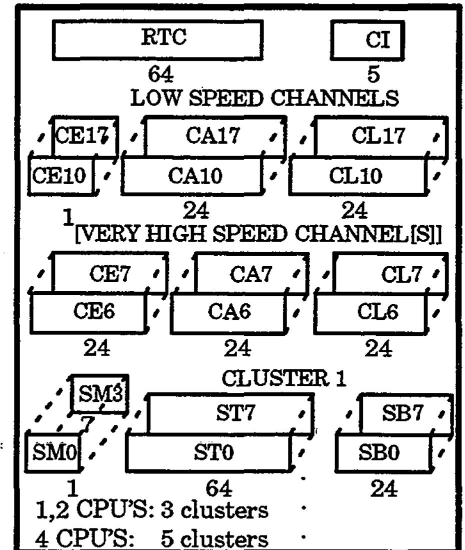
STATUS REGISTER BITS (073i01)

CL | PS FPS IFP IOR BDM PN CLN

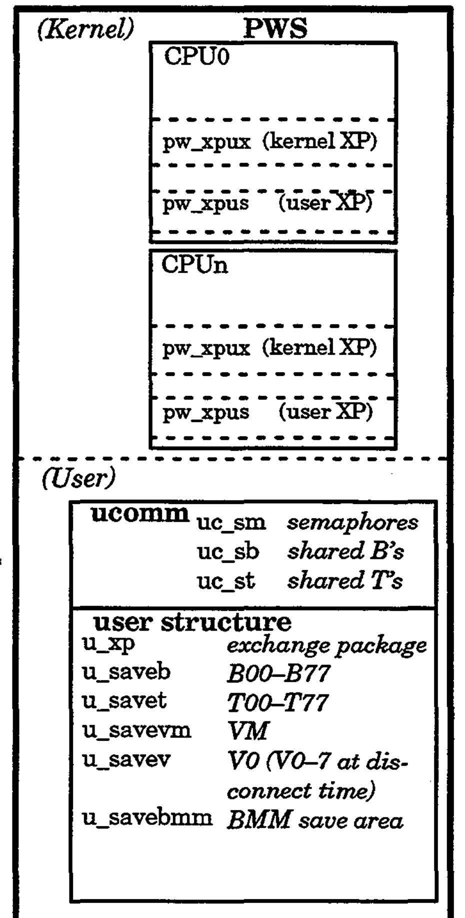
- CL – clustered (not in XP)
- PS – program state
- FPS – fl. pt. occurred
- IFP – fl. pt. enabled
- IOR – op. range enabled
- BDM – bidir. mem enabled
- PN – processor number (1,2 bits)
- CLN – cluster number (2,3 bits)

('Active Exchange Package contents')

OTHER SHARED RESOURCES



MEMORY



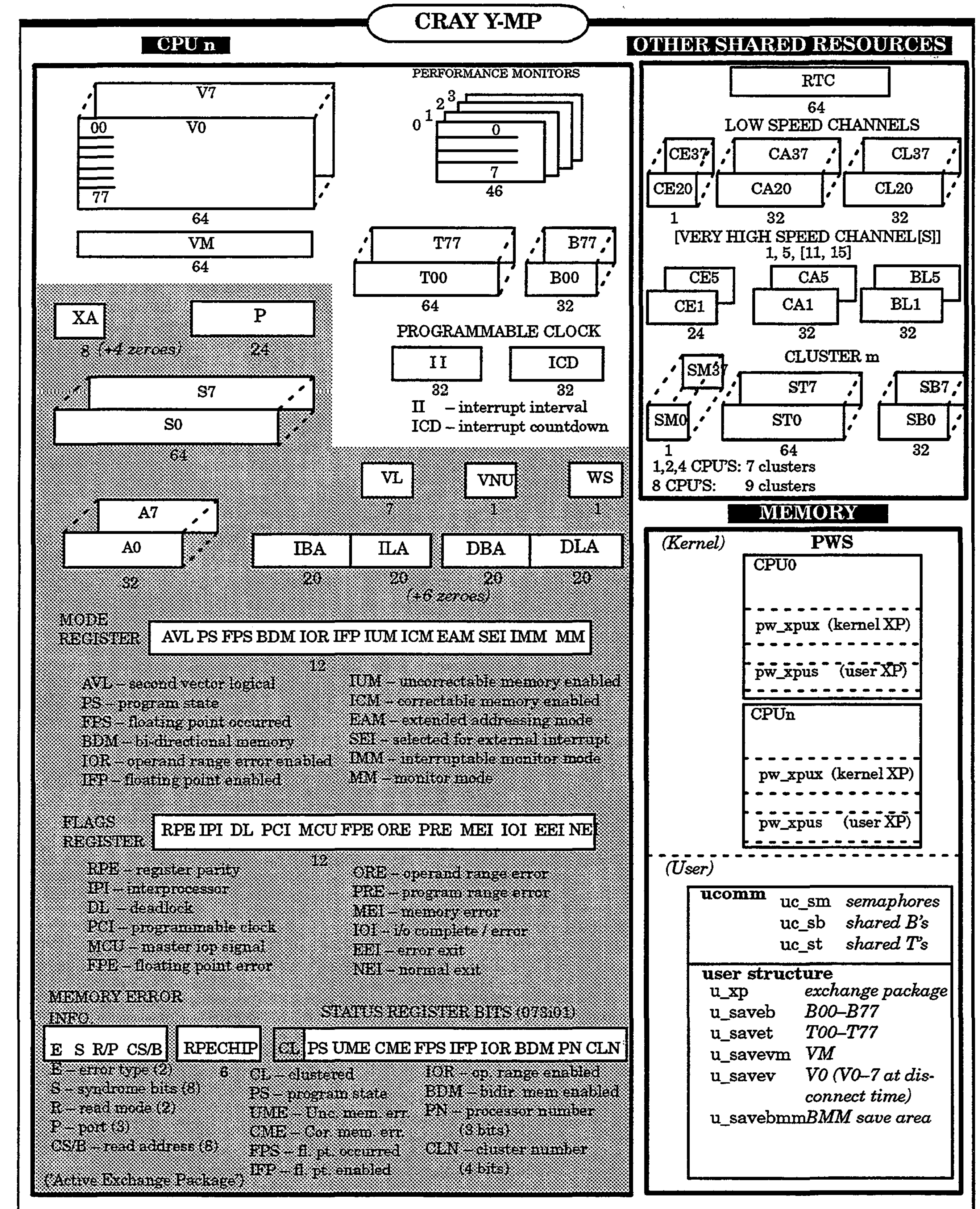
Interrupt Mode/Flag Summary (Y-MP C90)

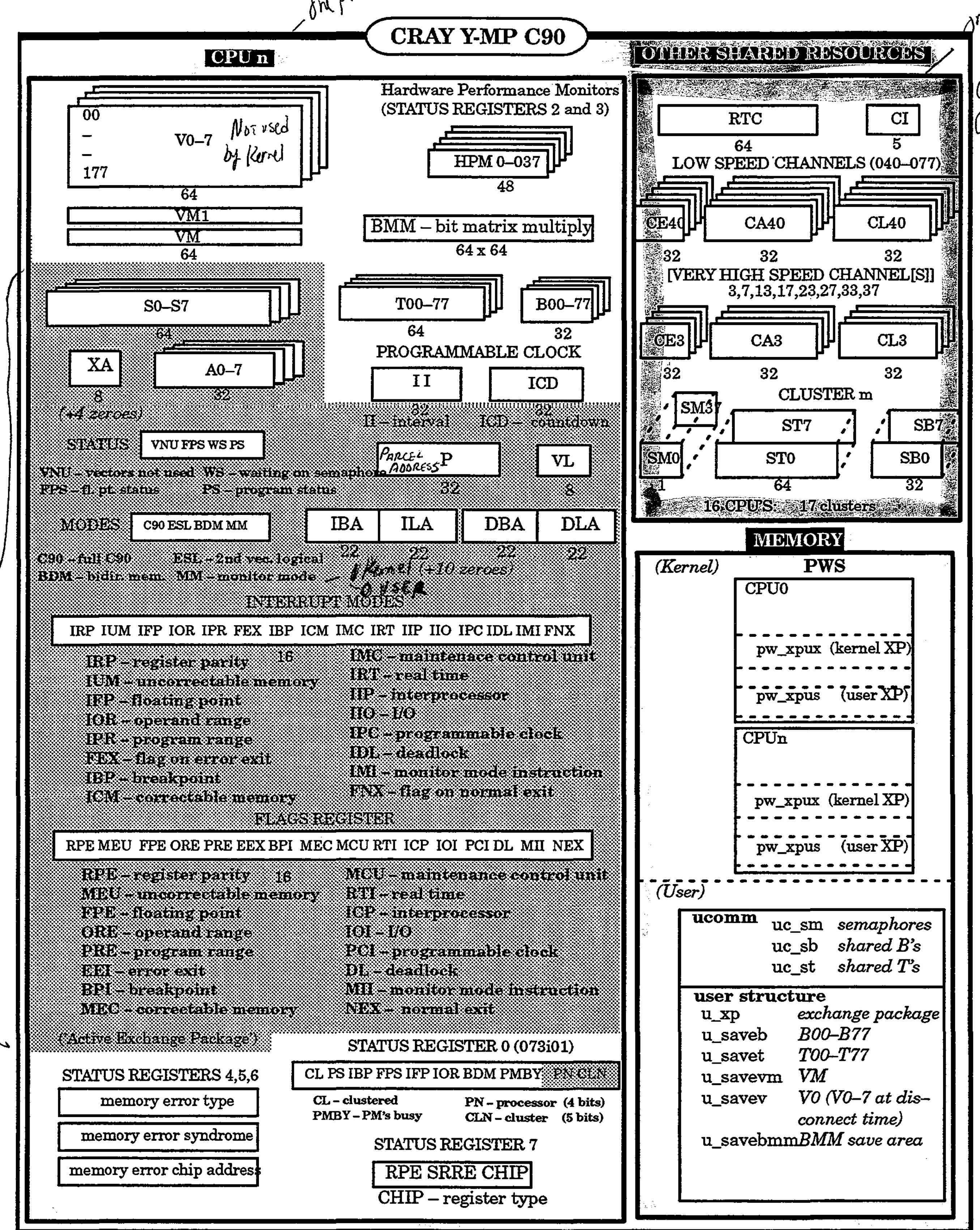
INT Flag (it happened) / Interrupt Mode (we care)

	Register parity **RPE / IRP**	Memory error uncorrectable **MEU / IUM**	Floating point error **FPE / IFP**	Operand range error **ORE / IOR**	Program range error **PRE / IPR**	Error exit **EEI / FEX**	Break point interrupt **BPI / IBP**	Correctable memory **MEC / ICM**
CPU in User (**mm=0**)	INT `usrrpe`	"?" if masked ignore else INT `usrmei`	"?" if masked ignore else INT `usrfpi`	"?" if masked ignore else INT `usrore`	Software says maskable, but hardware forces INT `usrpre`	INT `usreex`	INT `usrbpi`	"?" if masked ignore else INT `usrmei`
CPU in Kernel (**mm=1**)	INT `immrpe: panic` (unless scrubbing)	Software says maskable, but hardware forces INT `immmei: panic`	INT `immfpe: panic`	Software says maskable, but hardware forces INT `immore: panic`	Software says maskable, but hardware forces INT `immpre: panic`	INT `immeex: panic`	INT `immbpi: panic`	Masked (ignored) (if causes INT `immmei: panic`)

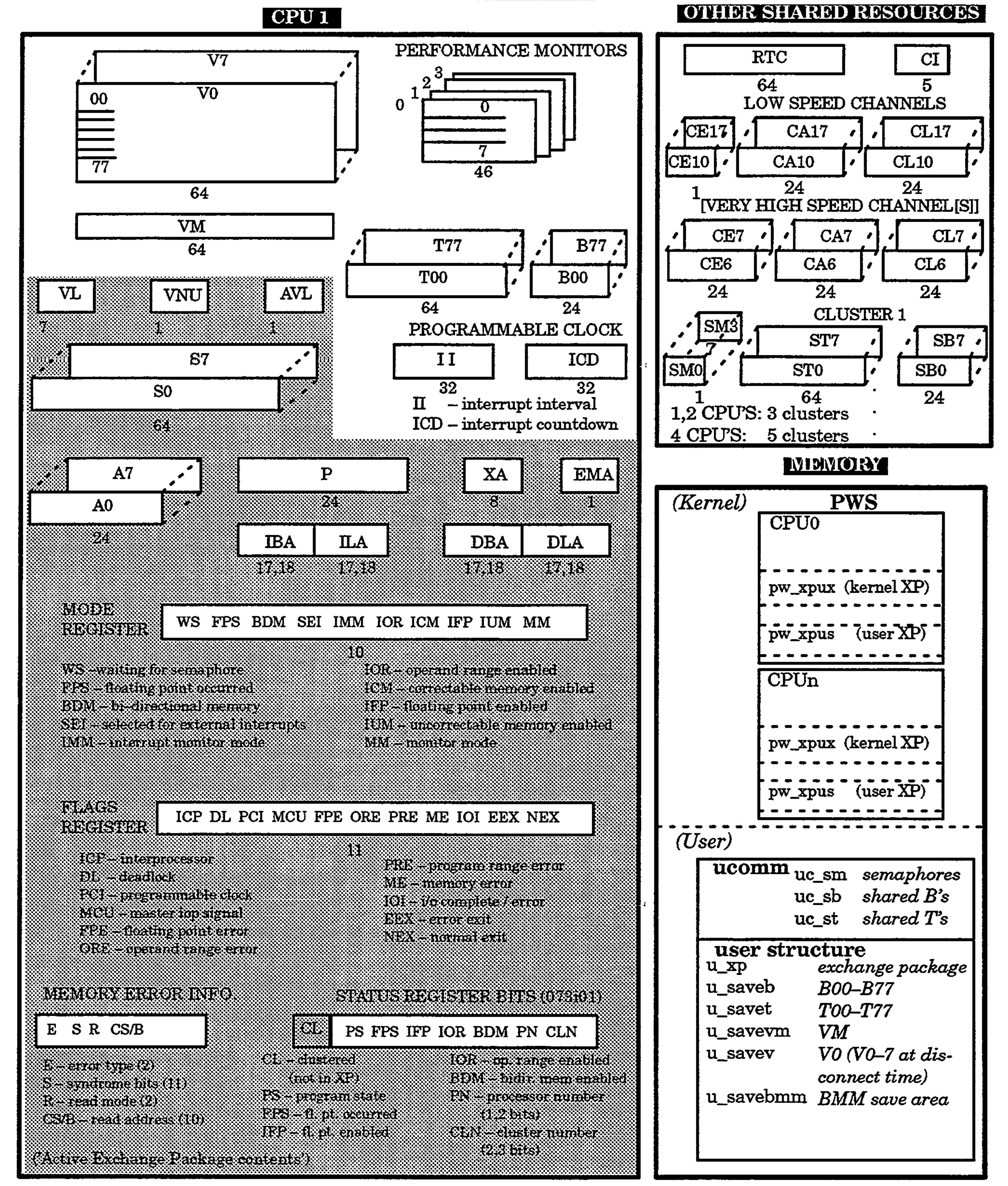
	Maintenance control unit **MCU / IMC**	Real time clock interrupt **RTI / IRT**	Inter-processor interrupt **ICP / IIP**	I/O interrupt **IOI / IIO**	Programmable clock INT **PCI / IPC**	Deadlock interrupt **DL(I) / IDL**	Monitor mode interrupt **MII / IMI**	Normal exchange **NEX / FNX**
CPU in User (**mm=0**)	INT `usrmcu`	INT `usrrtm` (same as `usrmcu`)	INT `usripi`	INT `usrioi`	INT `usrpci`	INT `usrdli`	INT `panic`	INT `usrnex` system call
CPU in Kernel (**mm=1**)	Pending (if causes INT `immmcu: panic`)	INT `immrtm: panic`	Pending (if causes INT `immipi: panic`)	Pending (if causes INT `immioi: panic`)	Pending (if causes INT `immpci: panic`)	INT `immdli: panic`	(if causes INT `immmii: panic`)	† INT `immnex: panic`

† Depends on XA: XA=pw_xpux ➧`immnex:panic`, XA=pw_xpus ➧user progam
XA=*other* ➧ idle loops, rpe scrub, diagnostic, memory error test, etc.


CRAY Y-MP
CPU n
OTHER SHARED RESOURCES
V7
00 V0
77
64
VM
64
PERFORMANCE MONITORS
0 1 2 3
0
7
46
T77
T00
64
B77
B00
32
XA
8 (+4 zeroes)
P
24
PROGRAMMABLE CLOCK
II
32
ICD
32
II – interrupt interval
ICD – interrupt countdown
S7
S0
64
VL
7
VNU
1
WS
1
A7
A0
32
IBA
20
ILA
20
DBA
20
DLA
20
(+6 zeroes)
MODE REGISTER
AVL PS FPS BDM IOR IFP IUM ICM EAM SEI IMM MM
12
AVL – second vector logical
PS – program state
FPS – floating point occurred
BDM – bi-directional memory
IOR – operand range error enabled
IFP – floating point enabled
IUM – uncorrectable memory enabled
ICM – correctable memory enabled
EAM – extended addressing mode
SEI – selected for external interrupt
IMM – interruptable monitor mode
MM – monitor mode
FLAGS REGISTER
RPE IPI DL PCI MCU FPE ORE PRE MEI IOI EEI NEI
12
RPE – register parity
IPI – interprocessor
DL – deadlock
PCI – programmable clock
MCU – master iop signal
FPE – floating point error
ORE – operand range error
PRE – program range error
MEI – memory error
IOI – i/o complete / error
EEI – error exit
NEI – normal exit
MEMORY ERROR INFO
E S R/P CS/B
RPECHIP
6
STATUS REGISTER BITS (07:01)
CL PS UME CME FPS IFP IOR BDM PN CLN
E – error type (2)
S – syndrome bits (8)
R – read mode (2)
P – port (3)
CS/B – read address (6)
CL – clustered
PS – program state
UME – Unc. mem. err.
CME – Cor. mem. err.
FPS – fl. pt. occurred
IFP – fl. pt. enabled
IOR – op. range enabled
BDM – bidir. mem enabled
PN – processor number (3 bits)
CLN – cluster number (4 bits)
(Active Exchange Package)
RTC
64
LOW SPEED CHANNELS
CE37
CE20
1
CA37
CA20
32
CL37
CL20
32
[VERY HIGH SPEED CHANNEL[S]]
1, 5, [11, 15]
CE5
CE1
24
CA5
CA1
32
BL5
BL1
32
CLUSTER m
SM37
SM0
1
ST7
ST0
64
SB7
SB0
32
1,2,4 CPU'S: 7 clusters
8 CPU'S: 9 clusters
MEMORY
(Kernel)
PWS
CPU0
pw_xpux (kernel XP)
pw_xpus (user XP)
CPUn
pw_xpux (kernel XP)
pw_xpus (user XP)
(User)
ucomm
uc_sm semaphores
uc_sb shared B's
uc_st shared T's
user structure
u_xp exchange package
u_saveb B00–B77
u_savet T00–T77
u_savevm VM
u_savev V0 (V0–7 at disconnect time)
u_savebmm BMM save area



CRAY Y-MP C90
CPU n
00 – – 177 V0–7
64
VM1
VM
64
Hardware Performance Monitors (STATUS REGISTERS 2 and 3)
HPM 0–037
48
BMM – bit matrix multiply
64 x 64
S0–S7
64
XA
8
A0–7
32
(+4 zeroes)
T00–77
64
B00–77
32
PROGRAMMABLE CLOCK
II
32
ICD
32
II – interval ICD – countdown
STATUS VNU FPS WS PS
VNU – vectors not used WS – waiting on semaphore
FPS – fl. pt. status PS – program status
P
32
VL
8
MODES C90 ESL BDM MM
C90 – full C90 ESL – 2nd vec. logical
BDM – bidir. mem. MM – monitor mode
IBA ILA DBA DLA
22 22 22 22
(+10 zeroes)
INTERRUPT MODES
IRP IUM IFP IOR IPR FEX IBP ICM IMC IRT IIP IIO IPC IDL IMI FNX
16
IRP – register parity
IUM – uncorrectable memory
IFP – floating point
IOR – operand range
IPR – program range
FEX – flag on error exit
IBP – breakpoint
ICM – correctable memory
IMC – maintenace control unit
IRT – real time
IIP – interprocessor
IIO – I/O
IPC – programmable clock
IDL – deadlock
IMI – monitor mode instruction
FNX – flag on normal exit
FLAGS REGISTER
RPE MEU FPE ORE PRE EEX BPI MEC MCU RTI ICP IOI PCI DL MII NEX
16
RPE – register parity
MEU – uncorrectable memory
FPE – floating point
ORE – operand range
PRE – program range
EEI – error exit
BPI – breakpoint
MEC – correctable memory
MCU – maintenance control unit
RTI – real time
ICP – interprocessor
IOI – I/O
PCI – programmable clock
DL – deadlock
MII – monitor mode instruction
NEX – normal exit
('Active Exchange Package')
STATUS REGISTER 0 (073i01)
CL PS IBP FPS IFP IOR BDM PMBY PN CLN
CL – clustered PN – processor (4 bits)
PMBY – PM's busy CLN – cluster (5 bits)
STATUS REGISTERS 4,5,6
memory error type
memory error syndrome
memory error chip address
STATUS REGISTER 7
RPE SRRE CHIP
CHIP – register type
OTHER SHARED RESOURCES
RTC
64
CI
5
LOW SPEED CHANNELS (040–077)
CE40 CA40 CL40
32 32 32
[VERY HIGH SPEED CHANNEL[S]]
3,7,13,17,23,27,33,37
CE3 CA3 CL3
32 32 32
CLUSTER m
SM3 ST7 SB7
SM0 ST0 SB0
1 64 32
16 CPUS 17 clusters
MEMORY
(Kernel) PWS
CPU0
pw_xpux (kernel XP)
pw_xpus (user XP)
CPUn
pw_xpux (kernel XP)
pw_xpus (user XP)
(User)
ucomm uc_sm semaphores
uc_sb shared B's
uc_st shared T's
user structure
u_xp exchange package
u_saveb B00–B77
u_savet T00–T77
u_savevm VM
u_savev V0 (V0–7 at dis-connect time)
u_savebmm BMM save area



Interrupt Mode/Flag Summary (Y-MP C90)

INT Flag (it happened) / Interrupt Mode (we care)

	Register parity **RPE / IRP**	Memory error uncorrectable **MEU / IUM**	Floating point error **FPE / IFP**	Operand range error **ORE / IOR**	Program range error **PRE / IPR**	Error exit **EEI / FEX**	Break point interrupt **BPI / IBP**	Correct-able memory **MEC / ICM**
CPU in User **(mm=0)**	INT `usrrpe`	"?" if masked ignore else INT `usrmei`	"?" if masked ignore else INT `usrfpi`	"?" if masked ignore else INT `usrore`	Software says maskable, but hardware forces INT `usrpre`	INT `usreex`	INT `usrbpi`	"?" if masked ignore else INT `usrmei`
CPU in Kernel **(mm=1)**	INT `immrpe: panic` (unless scrubbing)	Software says maskable, but hardware forces INT `immmei: panic`	INT `immfpe: panic`	Software says maskable, but hardware forces INT `immore: panic`	Software says maskable, but hardware forces INT `immpre: panic`	INT `immeex: panic`	INT `immbpi: panic`	Masked (ignored) (if causes INT `immmei: panic`)

	Maint-enance control unit **MCU / IMC**	Real time clock interrupt **RTI / IRT**	Inter-processor interrupt **ICP / IIP**	I/O interrupt **IOI / IIO**	Program-mable clock INT **PCI / IPC**	Deadlock interrupt **DL(I) / IDL**	Monitor mode interrupt **MII / IMI**	Normal exchange **NEX / FNX**
CPU in User **(mm=0)**	INT `usrmcu`	INT `usrrtm` (same as `usrmcu`)	INT `usripi`	INT `usrioi`	INT `usrpci`	INT `usrdli`	INT `panic` (Should never Get)	INT `usrnex` system call
CPU in Kernel **(mm=1)**	Pending (if causes INT `immmcu: panic`)	INT `immrtm: panic`	Pending (if causes INT `immipi: panic`)	Pending (if causes INT `immioi: panic`)	Pending (if causes INT `immpci: panic`)	INT `immdli: panic`	(if causes INT `immmii: panic`)	† INT `immnex: panic`

(if INT, we have problems should never happen

† Depends on XA: XA=pw_xpux ➧immnex:panic, XA=pw_xpus ➧user progam
XA=*other* ➧ idle loops, rpe scrub, diagnostic, memory error test, etc.

Contents

System Initialization [3] 3–1
Objectives 3–1
Overview 3–2
Kernel compile options 3–3
Kernel code optimization 3–3
Global register assignment 3–3
Global intrinsic functions 3–4
Vector use restrictions 3–5
Kernel mode intrinsic functions 3–5
Kernel mode intrinsic functions for CRAY Y-MP C90 systems 3–6
Assembler table macros 3–7
UNICOS linked lists 3–8
UNICOS kernel bit maps 3–10
UNICOS stacks 3–12
Stable stack feature summary 3–13
Stack pool management 3–14
stackinit() 3–14
expandstack() 3–14
contractstack() 3–14
Stack management 3–15
allocstack() 3–15
freestack() 3–15
Stack format 3–16
Context switching 3–18
CPU and process management 3–18
Basic principles 3–18
Kernel register save areas 3–18
Context switch sample 3–20
sleep() and wakeup() 3–22
Kernel main loop overview 3–24
Kernel multithreading 3–26
Overview 3–26
Lock mechanics 3–28
UNICOS multithread lock logic - general 3–29
SEMLOCK macro 3–30
SEMLOCK illustration 3–31
MEMLOCK macro 3–32
MEMLOCK illustration 3–33
ATOMIC lock macros 3–34
ATOMIC_ADD illustration 3–35
R_MEMLOCK and W_MEMLOCK lock macros 3–36
R_MEMLOCK and W_MEMLOCK illustration 3–39
Atomic sleep 3–40
Logic without atomic sleep 3–41
Logic with atomic sleep 3–41
Ownership macros 3–42

Intro stuff

New as of 8.0

Lock hierarchy 3–42
Lock statistics 3–43
Lock debugging 3–43
Kernel register uses 3–44
Kernel CPU register usage 3–44
Kernel cluster (1) register usage 3–44
Kernel cluster (1) semaphore register usage 3–45
Bootstrapping the mainframe 3–47
Booting methods 3–47
Bootstrapping the mainframe with the full kernel 3–48
Kernel structures at deadstart 3–50
Kernel structures at deadstart 3–51
Bootstrapping the mainframe with a compressed kernel 3–52
UNICOS kernel startup 3–55
Startup overview 3–55
`mfstart` / `mfinit` logic 3–56
`csl` processing 3–58
Startup file / table relocation 3–62
`umain()` logic 3–67
`sysproc()` routine 3–73
Summary 3–73
Creating system processes 3–73
`sysproc()` example 3–74
Creating system processes 3–76
Central memory sizes 3–82

ORIGINAL CHAPTER 3 STUFF

System Initialization [3]

Objectives

After completing this section you should be able to:

- **Describe kernel software features:**
  - **Compile options**
  - **Linked lists**
  - **Bit maps**
  - **Context switching**
  - **Mainline logic**
  - **Multithreading locks**
- **Describe the kernel deadstart procedure**
- **Trace the logic flow of the kernel from deadstart through `init`(8) in single user mode**
- **Diagram contents of memory during the various stages of startup**

Overview

The "System Initialization" chapter describes how the UNICOS kernel is started from a "down" system.

The first section of the chapter provides detail on several kernel software features providing background needed to understand kernel source code and logic.

The middle sections of the chapter describes the operational procedure relating to deadstarting the Cray mainframe, including special aspects of the files that are used to start the machine.

The third sections uses pseudo code / flow diagrams and memory layout diagrams to describe UNICOS startup processing.

Kernel compile options

This subsection describes special software coding practices used in the UNICOS kernel source code that are extensions to the typical C language programming environment. A unique C language compiler option and several assembler language programming facilities and conventions are used to support the UNICOS kernel. These software coding practices provide increased functionality within the kernel without sacrificing processing speed. They are created by compiling the UNICOS kernel. A general understanding of these unique coding practices provides a framework for understanding some of the essential functional characteristics of the kernel and the ability to study the logic of the kernel on a source code level.

The C compiler option `-h kernel` provides the following functional characteristics of the kernel:

- Supports a special set of intrinsic functions to allow the kernel access to hardware facilities beyond those accessible in the standard compiler
- Provides global register support
- Inhibits the compiler from generating vector code

The following subsections provide a detailed explanation of the `–h kernel` compiler option and how it affects the design of the kernel. A list of `–hkernel` invoked intrinsic functions is provided at the end of this subsection.

Assembler language coding conventions and table macros provide the following functional characteristics of the kernel:

- Allow for the referencing of C language structure members with assembler commands
- Provide a convenient method to access C language defined structure members with GET and PUT macros.

Kernel code optimization

The `-h kernel` command line option to the compiler provides additional optimization within kernel code. Separate returns are generated when the `-h kernel` option has been specified. Without the `-h kernel` option specified, all returns within a function jump to a compiler generated label within the function and this is where the return is actually done. When the `-h kernel` option is specified, each return within the function actually does the return and does not jump to the compiler-generated label.

Global register assignment

The UNIX kernel and UNICOS kernel make extensive use of global pointers. The term *global* in C language means that the data item is defined outside of the scope of a function block, has an external attribute, and therefore can be referenced by other functions linked in the same program without defining that value in each function itself.

However, it should be emphasized that "global" here does not imply accessible outside of the kernel or accessible outside a given process space. User-level processes do not have direct access to any global pointers that are in the domain of the kernel.

All of the values and tables defined in the file `/usr/src/uts/c1/md/lowmem.c` have this global attribute. Of particular importance to the kernel are four pointers normally defined in low memory (`lowmem`), which are pointers to (contain the address of) the current connected process' `proc` table area (up pointer), `pcomm` area (`upc` pointer), `user` area (u pointer), and `ucomm` area (uc pointer). Thousands of lines of kernel code reference fields within these four structures using syntax such as:

```
up->p_pflag
```

or

```
u->u_saveb
```

Global intrinsic functions

The concept of *current process* and the coding method described above conflict with the concept of the *multithreaded* kernel where there is a unique current process per CPU (up to 16 on the CRAY Y-MP C90). A single global pointer in low memory (`lowmem`) cannot support the multithreaded functionality of the kernel.

Global register intrinsic functions provide a way for a CPU-specific register to be used whenever a source statement references the memory management fields described above. CPU–specific work registers are normally selected from the set of `B01-B55` and `T01–T55` work registers. The global register intrinsic functions ensure that each CPU is referencing its own process area.

Global means only within the context of the kernel. The contents of these registers are not accessible to users. The kernel's contents in these registers are saved and replaced by the user's contents before the user is given access to these registers. While executing in user mode, a user process only has access to private data. The use of global register intrinsic functions provide no system integrity or security problems because they only provide convenient access to features that would otherwise require CAL programming. User mode use of these intrinsic functions does NOT provide the same semantics as kernel mode use.

The use of the `B01-B55` and `T01-T55` work registers by the kernel global register intrinsic functions causes some conflict with the normal assignment of these registers by the C compiler. To resolve this conflict, a special compile option must be used when building the kernel. This option requests that the compiler avoid using the selected registers for its normal work register pool. This command line option, `-h kernel`, is not documented for the field and is intended for internal Cray Research, Inc. use only.

A special syntax is recognized by the ANSI C front end to indicate that a variable must live in a fixed (global) `B` or `T` register. The variable type is limited to simple one-word data types equivalent to those allowed with the register attribute. The declaration looks like the following:

```
extern int GVAL  = _T(37);
```

or

```
extern int *PVAL = _B(25);
```

There is also a small restriction on the use of the B and T work registers. Registers must be assigned for global in the range of 1 through 55 decimal, 67 octal.

Source lines in the file /usr/src/uts/c1/sys/systm.h define the global variables and equate them to the work registers. The following external variable (extern) declarations define the registers assigned to pointers within major kernel structures for the current connected process as follows: the user area pointer (u) as CPU register B064 (octal), the ucomm area pointer (uc) as CPU register B063, the proc table pointer (up) as CPU register B062, and the pcomm area pointer (upc) as CPU register B061.

```
extern  struct  user  *u    =_B(064);

extern  struct  ucomm *uc   =_B(063);

extern  struct  proc  *up   =_B(062);

extern  struct  pcomm *upc  =_B(061);
```

In addition to these major global process pointers, a small number of B and T registers are assigned to variables to provide fast access to their corresponding data. The semaphore checking routines make use of most of these assigned registers.

In summary, a variable reference up->p_pflag in any function can be interpreted as _B(062)->p_pflag. Each CPU in the kernel thus has a unique reference to its own currently connected process (proc) table entries.

Vector use restrictions

As part of the kernel mode flag -h kernel machine code from the c compiler does not reference vector registers. Reference is avoided for the sake of kernel thread efficiency, since the time needed to save and reload vectors to preserve user data during interrupt processing usually exceeds any gains from other possible uses of vectors in kernel logic.

Vector registers are used in specific CAL routines such as memory to memory copies, but in general, it is up to the routine's author to preserve and restore the user's vector contents in these cases.

Kernel mode intrinsic functions

A number of special purpose intrinsic functions are available on the CRAY Y-MP family of computer systems. These functions are not documented for normal users on the system. Some of these intrinsic functions generate hardware privileged instructions such as providing the functioning of an I/O channel. The system must be executing in kernel mode to execute many of these intrinsic functions. Some examples of these intrinsic functions are as follows:

Function	Description
void_clrCI (*n*)	The _clrCI function clears the channel interrupt flag and channel error flag on channel *n*. No value is returned.
void _CCI ( )	The _CCI function clears the programmable clock interrupt. No value is returned.

Function	Description
`void_CIPI ( )`	The `_CIPI` function clears the interprocessor interrupt. No value is returned.
`void_ECI ( )`	The `_ECI` function enables the programmable clock interrupt. No value is returned.
`void _DCI ( )`	The `_DCI` function disables the programmable clock interrupt. No value is returned.

Kernel mode intrinsic functions for Cray Y-MP C90 computer systems

The following examples of intrinsic functions are available only on CRAY Y-MP C90 computer systems and only in kernel mode. These intrinsic functions are not documented for users.

Function	Description
`void _DI` (*n*)	The `_DI` function disables channel interrupts for channel *n*. No value is returned.
`void _DMI ( )`	The `_DMI` function disables monitor mode interrupts. No value is returned.
`void _EI` (*n*)	The `_EI` function enables channel interrupts for channel *n*. No value is returned.
`void _EMI ( )`	The `_EMI` function enables monitor mode interrupts. No value is returned.
`void _ESI ( )`	The `_ESI` function enables system I/O interrupts. No value is returned.

Assembler table macros

The assembler table creation macro `TABLE` is a standard UNICOS facility supplied and documented with the library routines. The source code for the macro `TABLE` resides in `/usr/src/lib/asdef/table.s`. The table consists of a header, body, and end statement. The body of the macro definition consists of `FIELD` macros. These `FIELD` macros provide for naming a particular field within a particular kernel table, indicate the field's bit starting position within a word, and the number of bits in the field. This field information is translated to a standard naming system using prefixes added to the field name supplied by a programmer. The three basic prefixes are:

Prefix	Description
W@	Word offset from the base of the table
N@	Bit offset in the word
S@	Size of the fields in bits

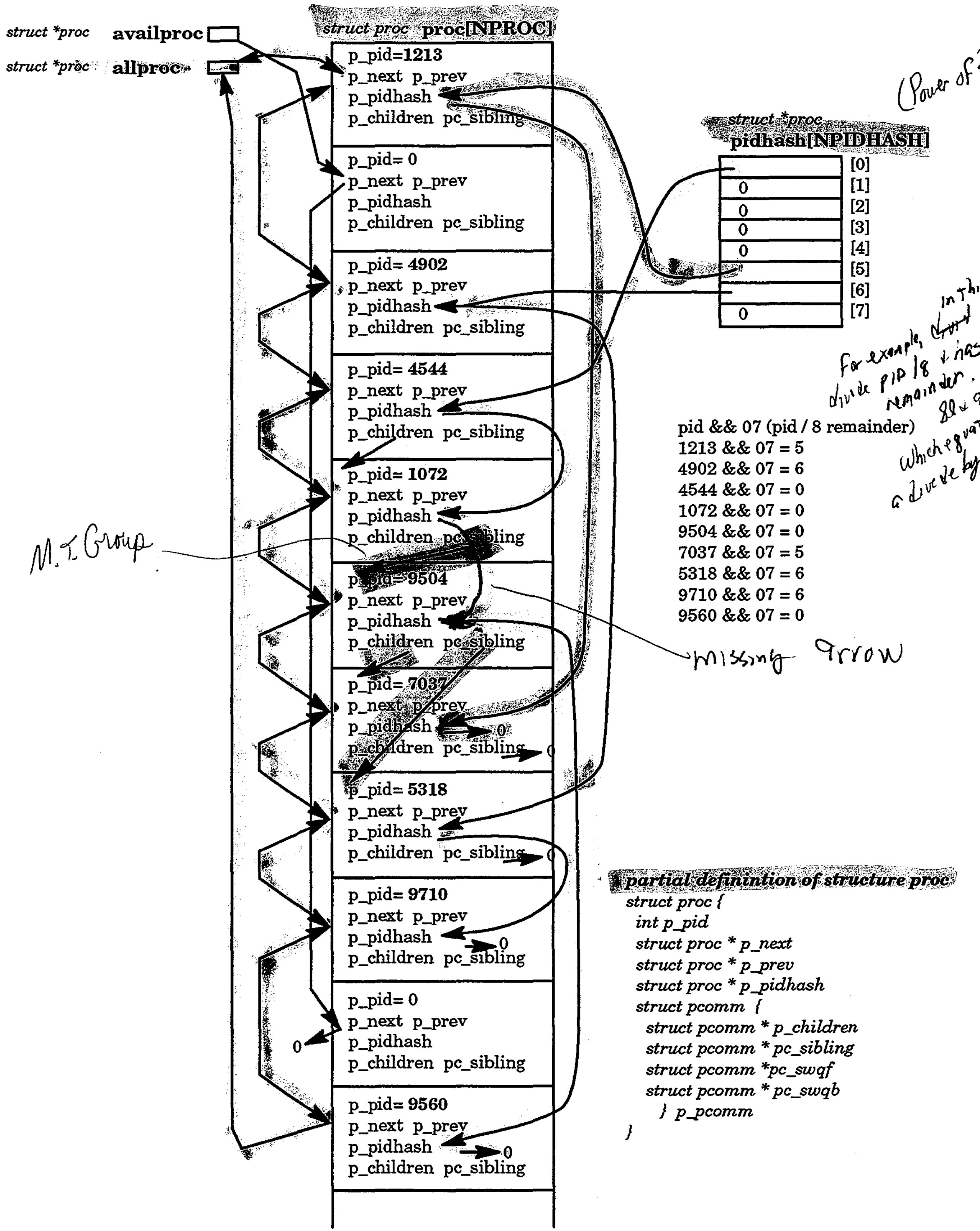
In general, kernel tables are not defined exclusively for assembler use but are defined by C language structure definitions in the standard header (`.h`) files. The assembler table macros define references to C structure fields so that the fields can only be accessed by the `$GET` and `$PUT` macros in a standard and convenient manner. Only table fields that are referenced by assembler code are defined in this way. A family of `GET` and `PUT` macros extract and insert bit field values when the field is referenced. The source code for the kernel macros is in the `/usr/src/lib/asdef` directory. The table references are defined in `cf.SN/utext.h`, which is created by the mkutext (`cmd/c1/mkutext/mkutext.c`) command compiled and executed during the kernel make process. To locate the C structure field references by the assembler W@ reference examine either the `utext.h` of `mkutext` files.

UNICOS linked lists

The UNICOS kernel makes extensive use of single and double linked lists.

- Single linked lists use a single address pointer to point to succeeding data items.
- Double linked lists link an item to its preceding and succeeding list item. These lists are used when the list is very dynamic, that is items are frequently added and removed from the middle of the list.
- UNICOS uses lists to form:
  - collections of "in-use" or "free" item's, (possibly table items).
  - queues of items indicating the items status.
  - queues of items indicating relationships (e.g. priorities or chronology).
  - "hash queues" to speed up table search time.
- Process table list examples (on right).

Type	List	Description	Sample
singly linked list	`availproc`	List of "free" process table entries – ones not assigned to a user's process. Members added and removed from the front only.	p_next points to succeeding elements – NULL terminated 2 entries (pid=0) on list
doubly linked list	`allproc`	List of process table entries assigned to user processes. Members added and removed anywhere in list.	p_next points to succeeding elements, p_prev points back – allproc is head and tail of the list pids 1213, 4902, 4544, etc. on list
queue	`p_children/ pc_sibling`	Singly linked list of process's current child processes.	p_children points to youngest child process, siblings (children of same parent) are linked via the pc_sibling NULL terminated list pid 4544 has 3 children, ids 1072, 9504, and 5318 pid 9504 has child 7037
sorted queue	`swapq`	Doubly linked list of process (pcomm) entries in descending order by swap in priority	(not shown) `pc_swqf` and `pc_swqb` form doubly linked list of pcomm proc table entries
hash queue(s)	`pidhash`	Hash lists for locating a process by its process id (pid). A hash header table is used to create smaller lists of processes to save search time. Each time a new process is created and a new pid is computed, its pid is hashed as shown, and the item linked to the corresponding hash queue. When searching for a particular pid in the future the target pid is hashed, and only the short hash list is scanned.	The sample shows the `proc` table hashed across 8 hash headers `pidhash`. The remainder when the pid is divided by the number of headers is used as an index to locate the hash list. Note, by using a power of 2 hash header table size, the remainder can be computed with a simple "and" binary operation, as shown.


struct *proc availproc
struct *proc allproc
struct proc proc[NPROC]
p_pid=1213
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 0
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 4902
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 4544
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 1072
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 9504
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 7037
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 5318
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 9710
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 0
p_next p_prev
p_pidhash
p_children pc_sibling
p_pid= 9560
p_next p_prev
p_pidhash
p_children pc_sibling
struct *proc
pidhash[NPIDHASH]
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
pid && 07 (pid / 8 remainder)
1213 && 07 = 5
4902 && 07 = 6
4544 && 07 = 0
1072 && 07 = 0
9504 && 07 = 0
7037 && 07 = 5
5318 && 07 = 6
9710 && 07 = 6
9560 && 07 = 0
partial definition of structure proc
struct proc {
int p_pid
struct proc * p_next
struct proc * p_prev
struct proc * p_pidhash
struct pcomm {
struct pcomm * p_children
struct pcomm * pc_sibling
struct pcomm *pc_swqf
struct pcomm * pc_swqb
} p_pcomm
}
(Power of 2
M.T. Group
missing arrow

UNICOS kernel bit maps

The kernel makes extensive use of bit maps to manage resources. Central (user) memory, swap space, file system space, and system buffers are examples of resources managed by bit maps.

The diagram on the right references `coremap` , the pool of central memory where user processes are maintained, showing the three major data areas involved in map management.

- The resource itself (starting at end of kernel, `lastaddr` in diagram).
- Structure `map` to manage the area (`coremap`).
- The bit map itself (`corebits`).

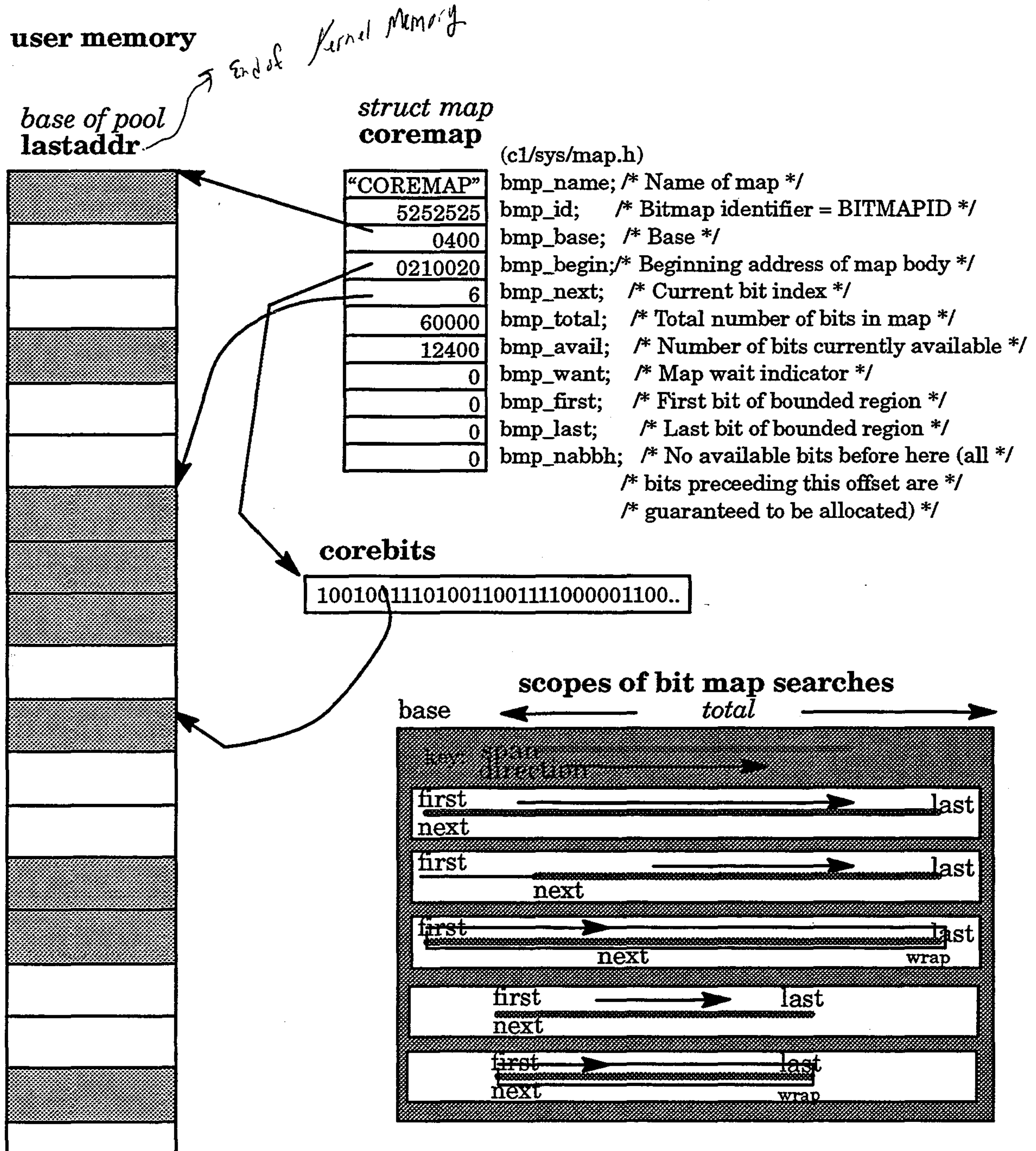
The "pool" itself can consist of arbitrary sized units, words, blocks, clicks, etc. There is one bit in the bit map for each unit. The position of the bit in the map corresponds to the position of the unit in the pool. The map routines assume a bit of "0" indicates a free unit and a bit of "1" is allocated. Structure `map` fields consist of:

Field	Description	Example
`bmp_name`	ASCII name of map	COREMAP
`bmp_id`	Validation number	BITMAPID = 05252525
`bmp_base`	"Address" of base of pool usually in the same unit as the pool itself	0400 (click address)
`bmp_begin`	Address of bit map itself	0210020
`bmp_next`	Allocate next units starting from this position	6
`bmp_total`	Size of entire pool (also number of bits in bit map)	60000
`bmp_avail`	Number of remaining units ("0" bits)	12400
`bmp_want`	Non-zero indicates another process is waiting for this resource	0
`bmp_first / bmp_last`	Option to bound allocation search to this range	0 / 0
`bmp_nabbh`	Indicates base of this area already entirely allocated	0

The kernel `malloc()` and `mfree()` functions (`c1/os/malloc.c`) are similar to user library routines provided to allocate heap space. The kernel `malloc()` function calls the assembler routine `mapget()` (`c1/md/bitmap.s`) to allocate the resource. Options arguments and options when calling `mapget()` include:

Field	Usage
map	The bit map accessed
num	Number of units requested
flag	Request options ("or'd" together) `M_ANY` Allocate from anywhere in map `M_HLD` Allocate from bmp_next only `M_NOWR` Do not wrap around at end of map `M_BEST` Return best effort if "num" can't be met `M_NEXT` Return next available set (after bmp_next) `M_EXACT` Return available block from closest fit `M_BOUND` Start search at bmp_first

mapget() returns the position of the first allocated unit if successful, or –1 on failure. Other map functions include: mapset() – set allocated bits on, mapret() – return units to the map, and mapsync() – report free unit in the map.



UNICOS stacks

Stacks are an integral part of C Language. UNICOS kernel stack processing characteristics:

- The kernel is compiled with the same C compiler that user applications use (perhaps an "older" version called a "generation compiler").
- The kernel logic uses stacks to save registers when calling and returning from kernel functions and to allocate function local variable data.
- The stack management routines used by the kernel are the same as those generated for user applications.
- Each process in UNICOS (including multitasked group members) has its own kernel stack. This stack is used to control function call logic and allocate local variables for the process as the kernel performs work on behalf of the process during interrupt processing.
- Special versions of stack processing functions, `setjmp()` and `longjmp()`, along with a number of other kernel specific functions, process kernel stacks as described later in this section.
- Previous to UNICOS release 8.0 kernel stacks were located in a fixed size area in the process's `user u_stack` area within the process's replaceable and swappable memory image.
  - On the "good" side, stack data was swapped out and released with the process memory.
  - On the "bad" side, data allocated on the stack had to be relocated when a process moved in memory (or was swapped out and in). This became a particularly big problem with the introduction of vnodes in 8.0 which placed many data items on the stack.
- Starting with UNICOS 8.0 stacks are assigned in a **stable stack** area. This stack pool area in high memory is reserved during startup. Each new process created is allocated an individual stack area from within this stack pool.
  - On the "good" side, stacks do not move once created, data placed on the stack does not need to be relocated when the process itself moves in memory.
  - On the "bad" side, stack space does not swap with the process, so stacks potentially take more memory than earlier release versions.

1380 is the stack size. It's as big as you can get.

Stable stack feature summary

- An initial stack pool is allocated in high memory during startup. This can grow (down in memory) to an upper limit size defined in the system.
- New processes created by `fork()` allocate a stack area for the kernel to use from this stack pool.
- When a CPU switches between processes (context switch), functions `setjmp()` and `longjmp()` save stack pointers for the "old" process and load stack pointers for the "new" process. The stack stays untouched in this stable memory area while the process is disconnected from the CPU.
- An option exists where the switch action can give up (deallocate) its stack when it is disconnected and reallocate it (as empty) when it is reconnected.
  - This is done in a few high frequency system call functions which follow this logic:

```
umain()
    sys_call_fun() {
        do initial housekeeping work
        request some system resource (e.g.. table entry)
        if resource not available
            request multiplexed stack
            set possible multiplexed stack function
            sleep(resource)
                swtch() disconnect CPU from process (setjmp() and longjmp() to
another
```

  - Normally the function would be resumed when reconnected at `sleep()` and continue on from there.
  - The "multiplex" indicates that the stack for this process can be given up (deleted) while the process sleeps (and is disconnected). The kernel will resume the process at `sys_call_fun()` when it is reconnected to the CPU after it is awakened.
  - The caller can request a specific function to execute before entering `sys_call_fun()` whose role would be to clean up any "loose ends" caused by this logic.

Detail on the stack area, the stack layout, and stack processing routines follows.

Stack pool management

The diagram on the right shows the kernel stack pool area in high memory. The following functions manage this stack pool area.

stackinit()

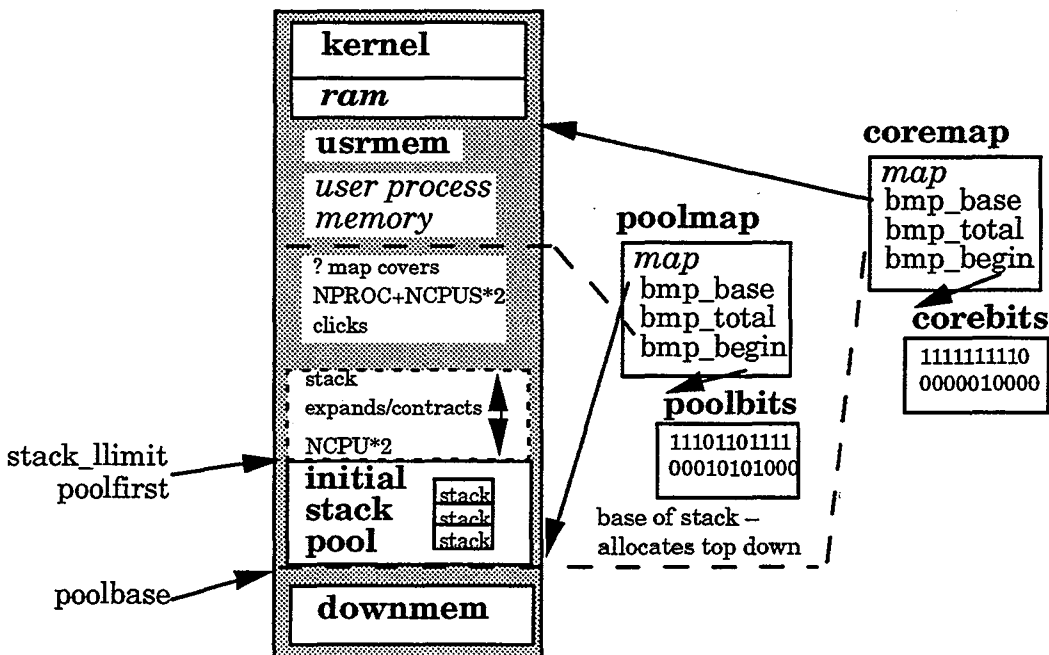
- c1/md/machdep.c
- Allocates initial stack pool at the end of user memory (coremap)
- Initial allocation (MAXCPUS+20) * NMPS each
  NMPS is 1380 words rounded up to MEMKLICK – the size of each usable stack area
- Initializes pool bitmap area
  - map management area called poolmap
  - Bitmap poolbits area big enough to grow pool to NPROC+NCPU*2
- poolbase is address last word of pool
- poolfirst is address of first word of pool within coremap (pool grows "downward" in memory, towards memory location 0)
- poollen is current pool length (in NMPS units)
- stack_llimit is first work of stack – absolute address
- usrmem (in clicks) decreased by poollen * NMPS
- downmem increased by poollen * NMPS

expandstack()

- c1/md/machdep.c
- Increases stack pool size by SEXPAND (NCPU * 2) units (each NMPS) allocated from coremap (note – stack expands downward in memory).
- Called each minor clock cycle (1/60 second) when the number of free stacks drops below NCPU*2 (see allocstack).
- If stack pool expansion fails because "that" area of memory in use, the swapper may be requested to shuffle processes (downward) in memory to create the space (detail in "sched" topic).
- poolfirst, stack_llimit, usrmem, and downmem adjusted to reflect new allocation.
- Calling process may sleep while waiting for expansion

contractstack()

- c1/md/machdep.c
- If less than a second since last contract, returns without effect.
- Releases SEXPAND (NCPU * 2) stack units – putting back in coremap and removing from poolmap.
- poolfirst, stack_llimit, usrmem, and downmem adjusted to reflect new allocation.



Stack management

Functions `allocstack()` and `freestack()` allocate and free stack areas within the stack pool. A stack is allocated when a new process is created by `fork()` or reallocated when a process is reconnected after giving up its stack for sleeping. A stack is freed when the process exits (leaves the system) or as an option when it gives up its stack during sleep.

`allocstack()`

- `c1/md/machdep.c`
- If the number of available stacks in the pool drops below `SEXPAND` (`NCPU*2`) set request `expandstack()` be executed at the next minor cycle.
- Allocate a stack area from stack pool. Map works from `poolbase` (high address) downward.
- If no stack space is available, the function sleeps or return with status (1) based on caller's option.
- Sets callers stack reference to newly allocated stack address (actual memory address)

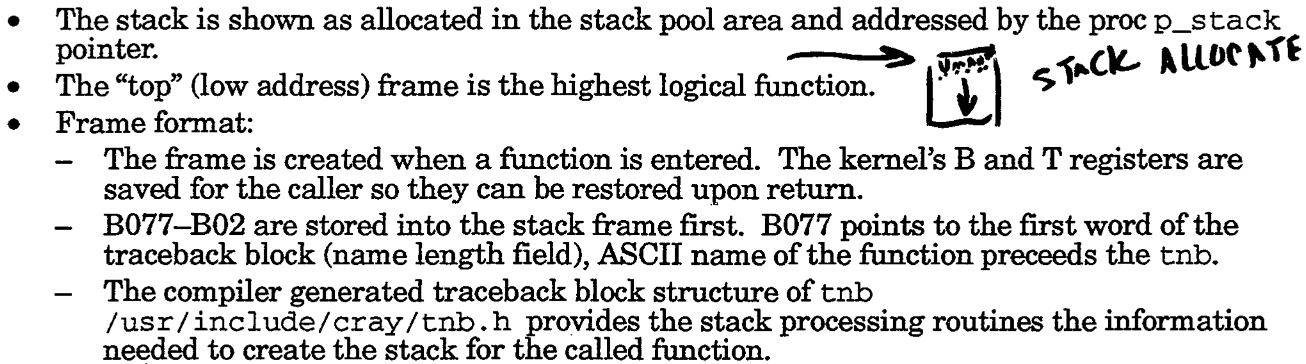
`freestack()`

- `c1/md/machdep.c`
- Frees one stack unit to stack pool.
- Clears calling processes reference to a stack (–1).

Stack format

The stack diagram on the facing page is built by a code sequence like:

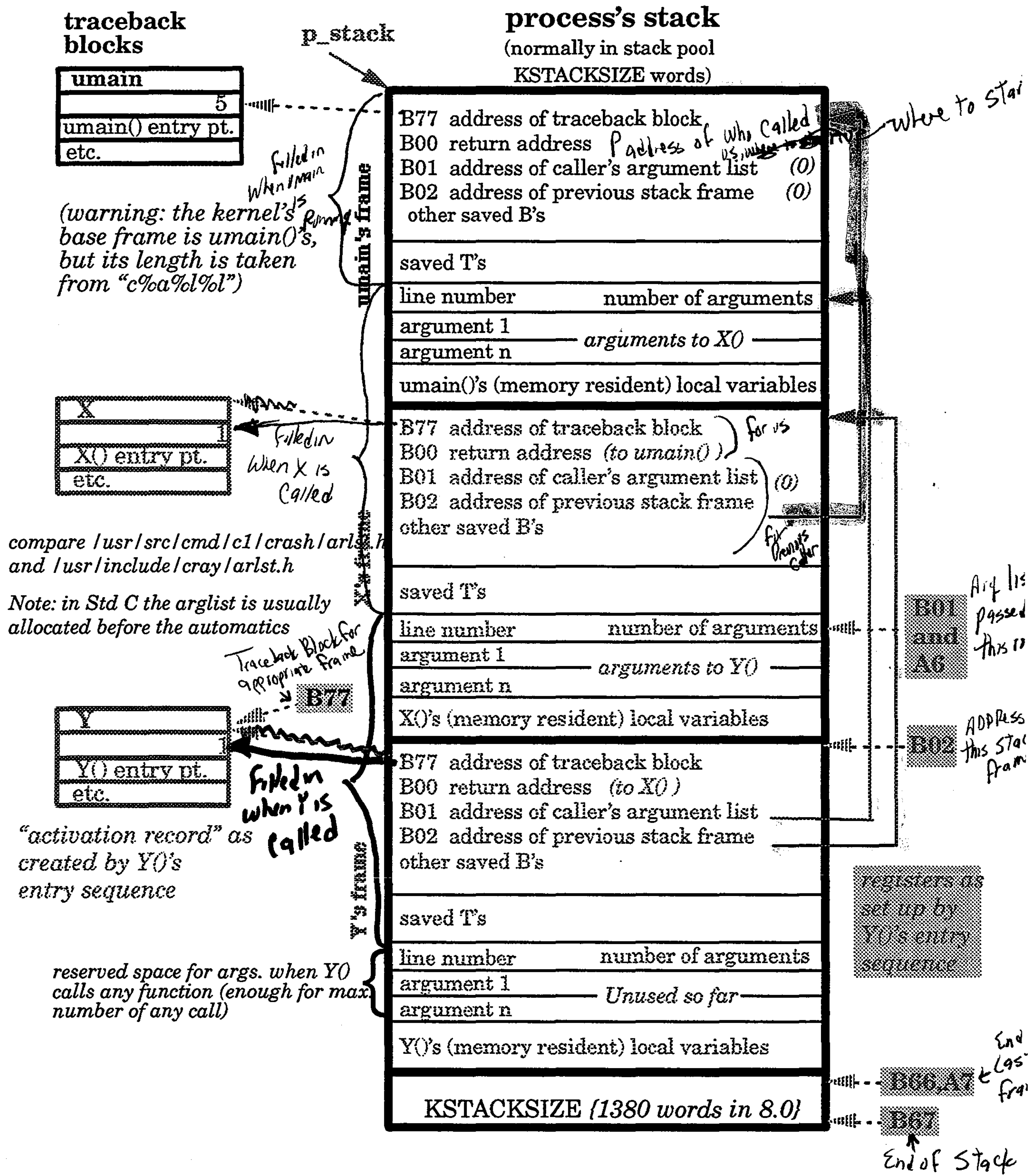
```
umain()
{ i = X(a, b); /* umain calls X */ }

X(i, j)
{ Y (q, r);  /* X calls Y */ }

Y(g, h)
{ /* stack shown in this context */ }
```



- The stack is shown as allocated in the stack pool area and addressed by the proc `p_stack` pointer.
- The "top" (low address) frame is the highest logical function.
- Frame format:
  - The frame is created when a function is entered. The kernel's B and T registers are saved for the caller so they can be restored upon return.
  - B077–B02 are stored into the stack frame first. B077 points to the first word of the traceback block (name length field), ASCII name of the function preceeds the `tnb`.
  - The compiler generated traceback block structure of `tnb` `/usr/include/cray/tnb.h` provides the stack processing routines the information needed to create the stack for the called function.

word	bits	field name
0	0–31	zero (0)
	32–47	tnbl (len traceback block)
	47–63	namel (len of name – char)
1	0–63	entrypt (function entry point)
2	0	base (level flag)
	1–7	lang (language type)
	8–19	argsize (max size arglist)
	20–31	NULL
	32–63	tvars (size of temp variable storage)
3	0–31	scons (size static constant storage)
	32–63	svars (size static variable storage)
4	0–49	NULL
	50–56	ntreg (number T-regs)
	57–73	nbreg (number B-regs)
5	0–63	langd (language dependent info)

  - The number of additional B's and T's is dependent upon how many are needed to preserve register-resident variables. Most register-resident variables are kept in A and S registers, but "overflow" into the B's and T's, or need to be preserved in B's or T's across a subroutine call.
  - The format of the argument list header differs between an X-MP and Y-MP. See the arlst.h header with the /etc/crash source for the Y-MP format.
  - The remainder of the frame is occupied by argument lists (built here for the call to other functions) and memory-resident variables. The compiler will try to make local variables register-resident, but structures and simple variables whose address is used (e.g. ptr = &variable) are allocated in memory.
- The active B01, B02, B066, B067, and B077 are shown in the context of executing function Y().



traceback blocks
p_stack
process's stack
(normally in stack pool KSTACKSIZE words)
umain
5
umain() entry pt.
etc.
B77 address of traceback block
B00 return address
B01 address of caller's argument list (0)
B02 address of previous stack frame (0)
other saved B's
(warning: the kernel's base frame is umain()'s, but its length is taken from "c%a%l%l")
umain's frame
saved T's
line number number of arguments
argument 1 arguments to X()
argument n
umain()'s (memory resident) local variables
X
1
X() entry pt.
etc.
B77 address of traceback block
B00 return address (to umain())
B01 address of caller's argument list (0)
B02 address of previous stack frame
other saved B's
compare /usr/src/cmd/c1/crash/arlst.h and /usr/include/cray/arlst.h
Note: in Std C the arglist is usually allocated before the automatics
X()'s frame
saved T's
line number number of arguments
argument 1 arguments to Y()
argument n
X()'s (memory resident) local variables
B01 and A6
B77
B02
Y
1
Y() entry pt.
etc.
B77 address of traceback block
B00 return address (to X())
B01 address of caller's argument list
B02 address of previous stack frame
other saved B's
"activation record" as created by Y()'s entry sequence
Y's frame
saved T's
registers as set up by Y()'s entry sequence
reserved space for args. when Y() calls any function (enough for max number of any call)
line number number of arguments
argument 1 Unused so far
argument n
Y()'s (memory resident) local variables
KSTACKSIZE {1380 words in 8.0}
B66,A7
B67

Context switching

CPU and process management

- There are usually many more user processes in the system than there are CPUs to service them.
- The CPUs switch between processes on a demand and priority basis.
- Switching must always be done when the CPU is executing the process "on behalf" of the user in the kernel.
- A **context switch** occurs when a CPU in the kernel disconnects from one process and connects to another.

Basic principles

The basic steps to perform when doing a context switch are:

- Disconnect "old" process.
  - Save all "user" register data in user save areas: XP with As Ss, Bs, Ts, Vs, CLs.
  - Save kernel work registers (Bs and Ts) in a kernel save area (also in user area).
  - Save the kernel stack for the process

Note: In UNICOS the stack is already "in memory", so all that needs saving are the stack B register pointers, which are saved as noted above).♦

- Select a new process to connect to (based on a priority scheme).
- Connect to the new process.
  - Restore kernel work registers (Bs and Ts) from the kernel save area.
  - Restore the kernel stack for the "new" process (note – actually part of next item).
  - Restore all "user" register data from user save areas: XP with As Ss, Bs, Ts, Vs, CLs.

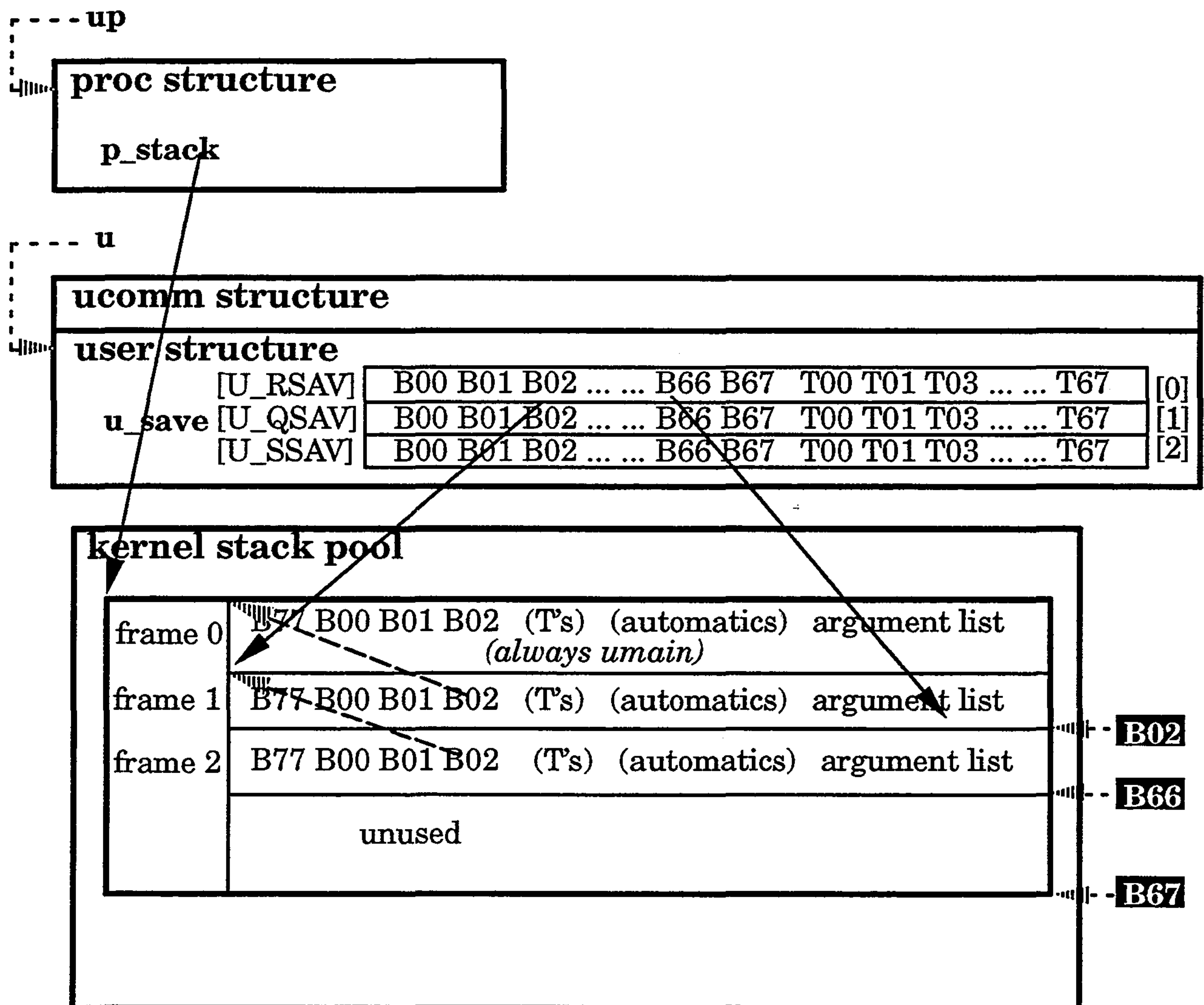
User register data is processed in the kernel mainline routine and discussed in the "Kernel Mainline" chapter. The following discussion relates to kernel register and stack processing during a context switch.

Kernel register save areas

The diagram on the facing page shows three different register save areas provided in a process's `user` area in the `u_save` array. Kernel defines are used to reference each as described below.

- `U_RSAV`: used by function `swtch()` to switch CPUs between processes as the normal part of CPU scheduling.
- `U_QSAV`: used for interruptable system call signal processing.
- `U_SSAV`: used when a CPU is disconnected for memory management reasons.

Kernel register and kernel stack saving and restoring by `setjmp()` and `longjmp()` are shown on the following pages.


up
proc structure
p_stack
u
ucomm structure
user structure
u_save
[U_RSAV] B00 B01 B02 ... ... B66 B67 T00 T01 T03 ... ... T67 [0]
[U_QSAV] B00 B01 B02 ... ... B66 B67 T00 T01 T03 ... ... T67 [1]
[U_SSAV] B00 B01 B02 ... ... B66 B67 T00 T01 T03 ... ... T67 [2]
kernel stack pool
frame 0 B77 B00 B01 B02 (T's) (automatics) argument list
(always umain)
frame 1 B77 B00 B01 B02 (T's) (automatics) argument list
frame 2 B77 B00 B01 B02 (T's) (automatics) argument list
unused
B02
B66
B67

Context switch sample

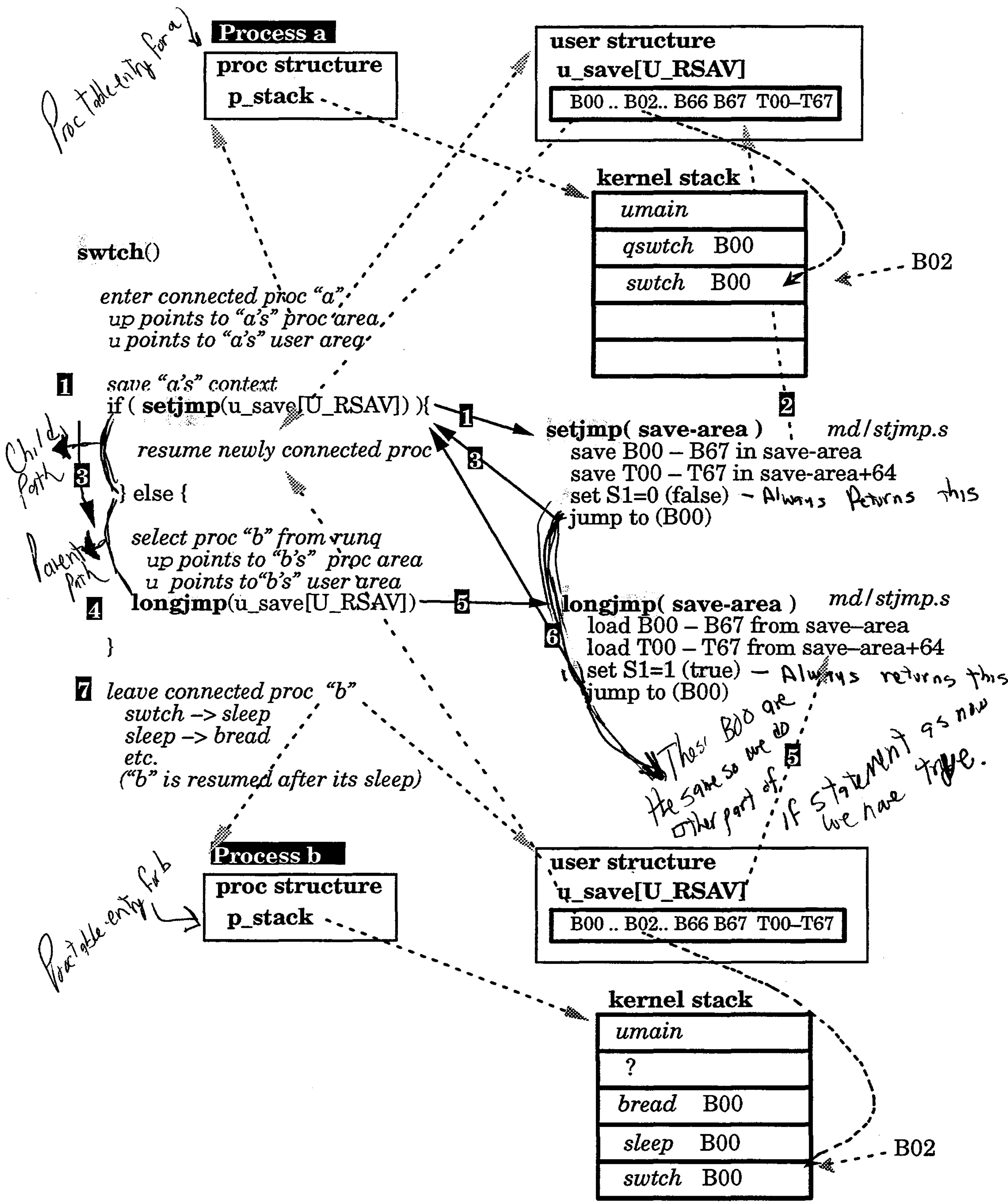
The diagram on the right illustrates how **setjmp()** and **longjmp()** are used in context switching to save register and stack contents. This example makes the following assumptions:

- Processing starts with process "a" connected and calling swtch() called by qswtch() (part of normal CPU scheduling). Note "a's" stack.
- Process "b" is disconnected (in the manner that is shown for "a") by a read calling sleep() calling swtch() (note its stack).

The context switch takes place in the following way:

1. Process "a", in swtch(), calls setjmp(). Setjmp() is implemented as a CAL routine, no new stack frame is created when it is entered.
2. Setjmp() saves the kernel B and T registers (0–67) into the specified ("a's") user save area. Register S1 (the function return value register) is set to zero (0).
3. Setjmp() returns to caller, the if statement. The return value of zero from setjmp() causes logic to take the "false" path.
4. Swtch() selects a new process to run from the run queue – we'll assume it's "b" – and sets its global pointers to reference "b's" proc and user areas (thus its stack and stack save area).
5. Swtch() calls longjmp(). Longjmp() is also a CAL routine, entering it creates no new stack frame. Longjmp() loads "b's" B and T registers from the specified save area. Note that the CPU's B00 gets set to the value B00 at the time "b" was disconnected in the past. The return value register S1 is set to 1.
6. When longjmp() "returns" it jumps to the instruction after the call to setjmp() – in the if statement. The "true path" of code in swtch() resumes the "new" process by performing any "housekeeping" action required to get it started again (for example, relocate its BA/LA regs).
7. When swtch() returns it is now connected to "b". The CPU returns to sleep() which returns to bread(), and eventually exchanges back to "b's" user program.

At some future time the disconnected process "a" will be selected by swtch() to be reconnected. "A" will be resumed in the same fashion as was shown for "b" in the example.


Process a
proc structure
p_stack
user structure
u_save[U_RSAV]
B00 .. B02.. B66 B67 T00–T67
kernel stack
umain
qswtch B00
swtch B00
B02
swtch()
enter connected proc "a"
up points to "a's" proc area
u points to "a's" user area
1
save "a's" context
if (setjmp(u_save[U_RSAV])){
resume newly connected proc
3
} else {
select proc "b" from runq
up points to "b's" proc area
u points to "b's" user area
4
longjmp(u_save[U_RSAV])
5
}
7
leave connected proc "b"
swtch –> sleep
sleep –> bread
etc.
("b" is resumed after its sleep)
2
setjmp(save-area) md/stjmp.s
save B00 – B67 in save-area
save T00 – T67 in save-area+64
set S1=0 (false)
jump to (B00)
longjmp(save-area) md/stjmp.s
load B00 – B67 from save–area
load T00 – T67 from save–area+64
set S1=1 (true)
jump to (B00)
6
Process b
proc structure
p_stack
user structure
u_save[U_RSAV]
B00 .. B02.. B66 B67 T00–T67
kernel stack
umain
?
bread B00
sleep B00
swtch B00
B02

sleep() and wakeup()

Function sleep() is called when a process must wait for a system resource or system event. It voluntarily gives up its CPU while waiting.

Function wakeup() is executed by a non-sleeping (runnable and connected) process when the resource becomes available or the event occurs. Wakeup makes the sleeping process runnable so it can (and will) be reconnected when normal CPU context switch action (swtch) occurs.

Processes that are runnable are on the run queue runq. Sleeping processes are queued on a set of hash queues known generically as the "sleep queue." In general sleep() removes a process from the runq and links it to the sleep queue. wakeup() removes a process from the sleep queue and links it to the runq. The runq is sorted by process priority with 0 (best) in front. The sleep queues are managed generally as FIFO (oldest first) queues.

The code segments on the facing page illustrate the most basic operation of sleep/wakeup. The logic follows these steps:

1. Process "a" attempts to allocate a table entry from a free list. *Assume the list is currently empty.*
2. Process "a" indicates it "wants" an item on the free list
3. Process "a" calls sleep() to give up the CPU until an entry becomes available. Note the process calls sleep() with 2 arguments:
   - freelist: the address of a data item that represents the resource the process "is sleeping on."
   - sleep_pri: sleep priority (controls type of sleep).
4. Funtion sleep():
   - removes the process from the runq and links it to a sleep hash queue.
   - calls swtch() which :
     - disconnects the process.
     - searches the runq for another process ("b").
     - connects the CPU to the other process.

   At this point process "a" is **NOT** runnable and will not be selected for reconnection by swtch() until it becomes runnable (on the runq).
5. Sometime in the future another process ("b") no longer needs its table item and links it to the freelist. After doing so it is "responsible" for testing if another process is "sleeping on" "that" resource. If so ...
6. it calls wakeup() to make the sleeping process runnable. Wakeup is called with the address of the resource what the process would be sleeping on. The sleep_pri value control where "a" will be placed on the runq.
7. Process "a" is runnable, but must wait until a CPU becomes available (e.g. another process call swtch() and "a" has the best priority (position on the runq)
8. Eventually "a" gets selected by swtch(). The CPU doing this does not have to be the same one that was connected to "a" earlier.
9. The "wanted" flag is cleared and the table item can be allocated by "a".
10. Process "a" continues processing in CPU "y".

Other features related to sleep() and wakeup() are covered in detail in Chapter 5's "Process Management Subsystem" section.

process "a" connected to CPU "x"

```
while ( freelist == NULL ) {        [1]
    wanted++;                       [2]
    sleep(freelist,sleep_pri);      [3]
    }
wanted--;  [9]
/* allocate entry from the freelist and continue */
   [10]
```

[4] **process "a" sleeping, swtch() disconnects CPU and connects to another process ("b")**

process "a" resumes processing in CPU "y"

[8]

```
    /* finish processing entry and link to the free list */
[5] if (wanted) {
[6]     wakeup(freelist);
    }
```

[7] **process "a" now runnable, it waits for a CPU to select it and connect via swtch()**

Kernel main loop overview

The diagram on the right represents the entire system – both kernel and user processes. Kernel initialization is pictured near the left, from **mfstart() through sched()** for CPU 0, and starting at park() for the other CPUs. It is only executed once, as each CPU heads for the "master" loop.

The diagram illustrates the logical path a CPU takes when it enters an idle process, and the path CPU 0 takes to resume the initialization logic after each I/O interrupt.

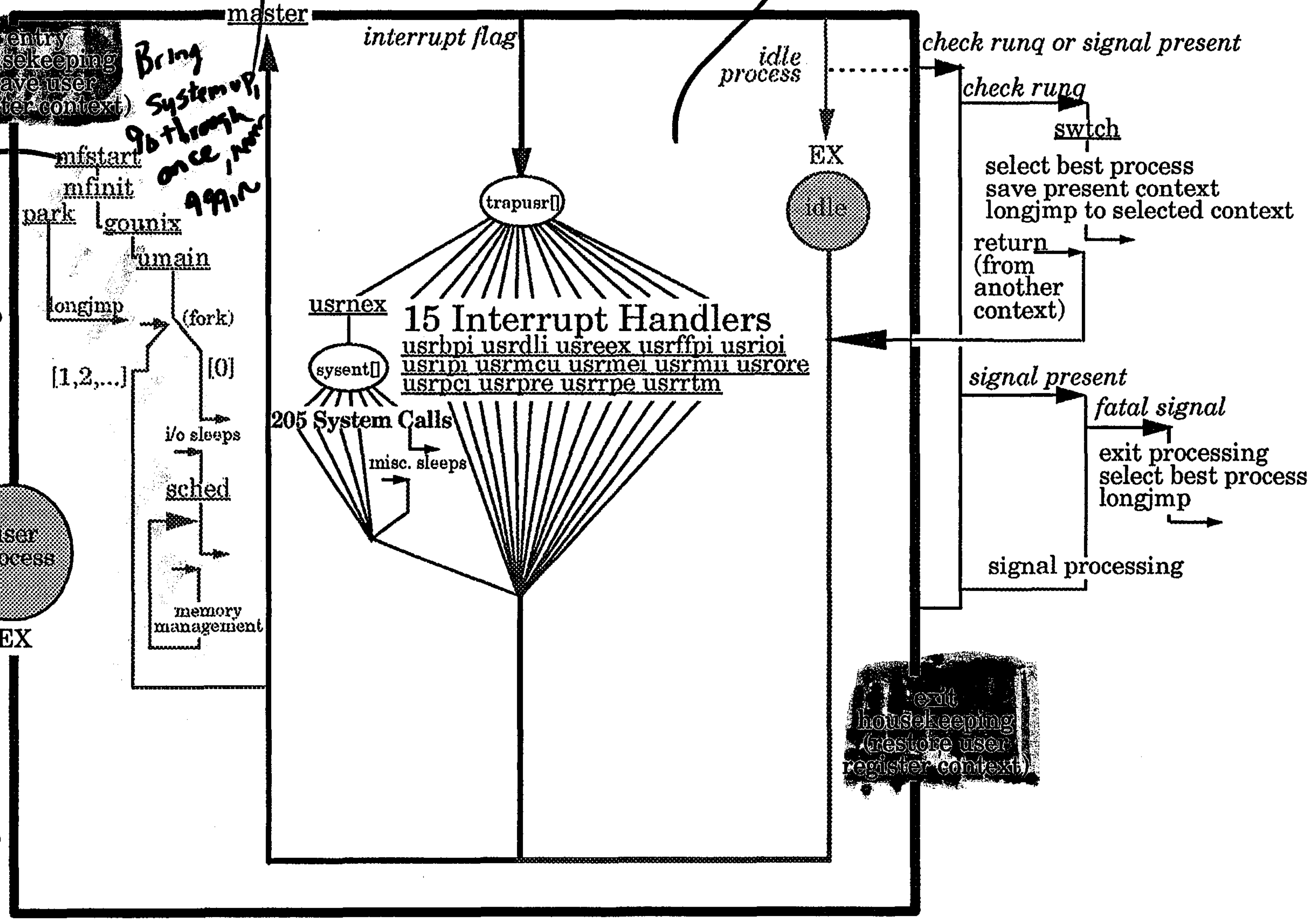
For a full, detailed diagram of the kernel's main loop see chapter 4.

(The arrows pointing off to "anywhere" represent context switches to other processes. The arrows coming back from "anywhere" represent context switches back to a logic thread.)

CPUs other than 0 will enter the context of their idle processes when they are created and spin wait there until kernel initialization is complete.

The entire initialization thread from mfstart() through sched() is executed in **the context of process[0]**. The sched() function becomes the only work performed in the context of process[0] after initialization is complete.

Kernel main loop overview



Kernel multithreading

Overview

The multithreaded UNICOS kernel uses hardware semaphores combined with a comprehensive set of software macros to protect memory areas shared among CPUs from being simultaneously updated.

- Most of kernel memory tables and work fields are in common memory and global in nature, any CPU can reference and change these values.
- Two basic types of locks are provided to protect data from simultaneous update by more than one CPU (see the figure on the right).
  - The SEMLOCK macro provides course grained locking , for example locking the whole process table during a process queue (linked list) update. The term "semaphore lock" is used for this type.
  - The MEMLOCK macro provides fine grained locking, for example locking a single process table entry while the kernel is updating its fields. The term "memory lock" is used for this type.
- Atomic locks are memory lock variants used to protect a field during a single operation, for example while adding to an individual process table count field.
- Macros R_MEMLOCK and W_MEMLOCK are memory lock variants used to provide multiple reader / single writer memory locks.

Nmakefile *uts/cf/Nmakefile*

The following lines of the kernel's Nmakefile file control kernel multi-threading:

```
#if productline(cray1)
...
MULTI_THREADING =    1
/* SEMDEBUG =    1    /* Uncomment to turn on SEMDEBUG */
/* SEMLOCKRULE   =    1    /* Uncomment to compile SEMLOCKRULE */
/* SEMTIMING     =    1    /* Uncomment to compile SEMTIMING */
#endif
```

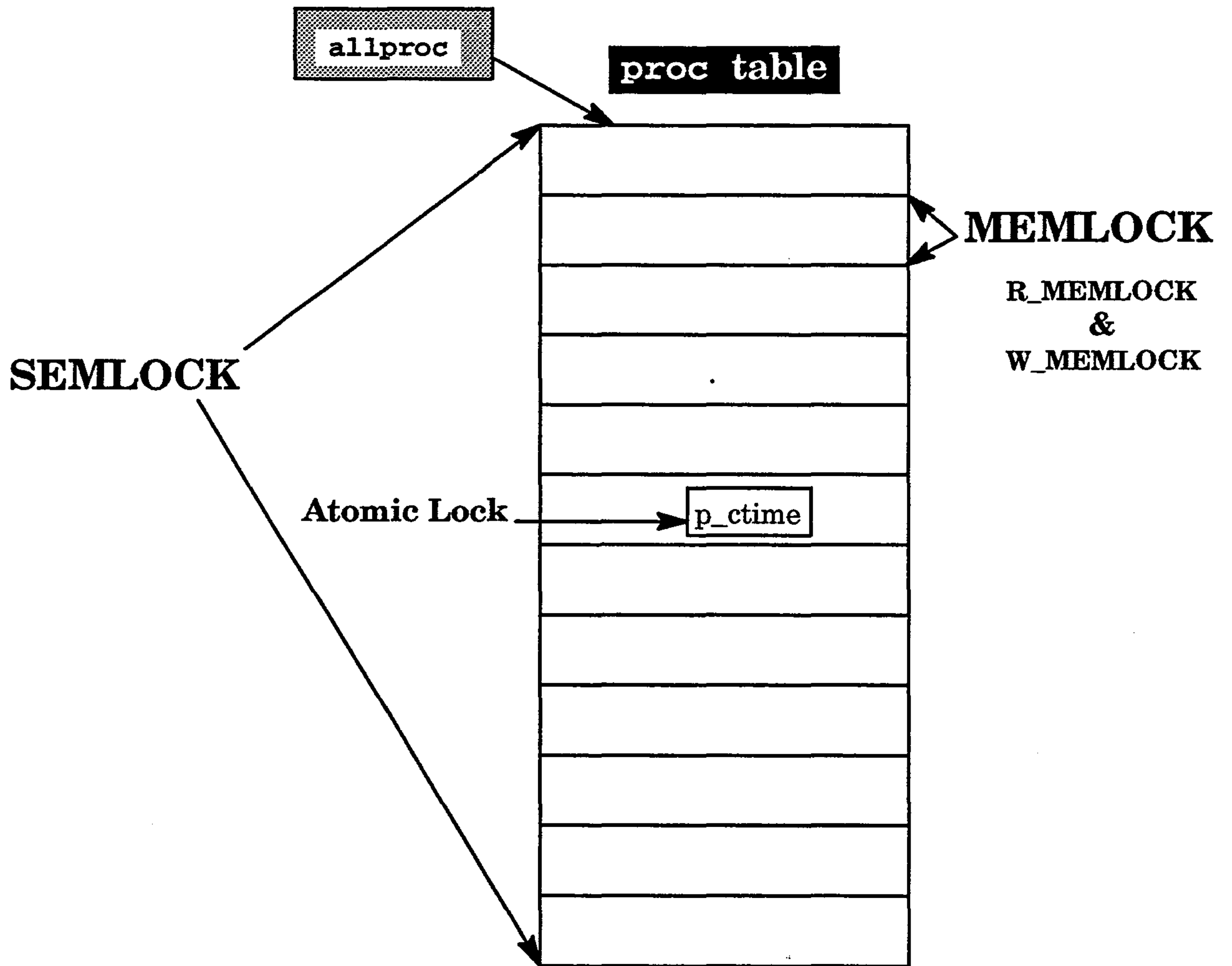
The MULTI_THREADING value "1" indicates that 8.0 multi-threading (as documented in this section) should be built into the kernel. MULTI_THREADING = 0 indicates the semaphore locks used prior to 8.0 are built into the kernel (called "single threading", but actually not). Kernels built for single CPU systems have neither type of locks active (ifdef'd out).

The three "SEM" values control collection of debugging and timing information. These are described at the end of this material.

The option to build and run a kernel with 8.0 multi-threading "off" is for development purposes only, it has not been tested on production systems.

Adding debugging and timing to production systems is not recommended for performance reasons (see description at the end of this material).

Kernel Locking Macros



Lock mechanics

- All CPUs executing kernel code reference hardware cluster 1 by UNICOS convention. The hardware semaphores and shared registers in this cluster are used to implement kernel multithreading locks.
- Hard locks - the test-and-set instruction `TS`
  - Logic
    ```
    SMn  1,TS   test and set SMn
         while SMn != 0
              (spin wait)
         set SMn = 1
    Do single threaded logic
    SMn  0    clear SMn
    ```
  - CPU "race conditions" are properly dealt with since the internal hardware logic of `TS` guarantees that only one CPU can interrogate and change the value in the semaphore at one time.
- Soft locks

  Certain limitations of the test and set instruction demand that an additional mechanism is used with the `TS` "hard lock".

  - Test and set does not provide control over which CPU is released as the lock is cleared by the "owning" CPU. (It is not a FIFO queue).
  - Soft lock code can provide performance statistics on locks.
  - Soft lock logic uses memory areas along with cluster 1 semaphores and shared registers to provide the locking mechanism.

Hardlock will protect a soft lock

Reference Pages, not covered. Text on Left, Example on Right

UNICOS multithread lock logic - general

- Testing and setting of lock (beginning of thread)

  In the following, assume cluster 1 semaphore 26 (`ISEMA`) is used to guard lock manipulation, and semaphore `sema`, shared T register 3 (`SEMMASK`), and `semowner[sema]` are used to protect "this" specific segment of code and the data it accesses.

Note: Semaphore timing statistics gathering is site selectable. ♦

```
LOCK(sema)
   TS  ISEMA  get general hard lock
   if sema != 0  test specific hard lock
      /* wait for thread lock */
      wstart=RTC start sema wait timer
      FIFO(end) -> CPU place CPU on FIFO queue this sem
      SEMMASK | CPU_mask mark ST SEMMASK CPU bit
      ISEMA  0 clear general hard lock
      while (SEMMASK CPU bit != 0)
         spin wait (8*no_locked_CPUs) clocks
      /* CPU has resumed after lock cleared by another CPU (below) */
      FIFO(start) = 0  remove CPU from sema FIFO queue
      wait+=RTC-wstart add to CPU/sema wait time total
      ISEMA  0 clear general hard lock
    else
      /* claim thread lock */
      TS  sema set specific hard lock
      ISEMA  0 clear general hard lock
   semowner[sema] | CPU_mask  mark this CPU "owns" sema
   tstart=RTC begin single thread timer
```

- Clearing Lock (at end of thread)

```
UNLOCK(sema)
   time+=RTC-tstart  end single thread timer
   TS  ISEMA set general hard lock
   semowner[sema] & !CPU_mask  clear owning CPU's bit
   if FIFO != 0 CPU waiting for this sem
      /* select and start "oldest" CPU */
      CPUn = FIFO  select CPU at head of queue
      SEMMASK & ! CPU_mask      clear ST SEMMASK bit CPUn
      /* selected CPU will drop out of spin
         loop - it inherits locked semaphores */
    else
      /* no CPUs waiting - clear
         semaphores and continue */
      sema 0  clear specific hard lock
      ISEMA  0  clear general hard lock
   /* "this CPU continues on CPUn can proceed as lock owner */
```

SEMLOCK macro

The SEMLOCK macro is used to test for and set a course-grained lock on a data area. For example, it would be used to protect a linked list or queue while the list is being updated. Since these locks may be held longer in time and are broader in nature than MEMLOCK memory locks, they are used sparingly in the kernel.

- SEMLOCK macro `include/sys/semmacros.h`
  `#define SEMLOCK(sema, lid)`

`sema`	Semaphore number - used by this instance of the call
`lid`	Lock id - unique number used as index to timing and lock rule tables

- A full set of macros defined in `include/sys/semmacros.h` reference SEMLOCK with proper `sema` and `lid` UNICOS kernel values. Example:
  `#define BUFTAB_LOCK()          SEMLOCK(BUFLOCK, BUFLOCK_LID)`
- The diagram on the right shows key memory areas and illustrates how they are used in processing a lock on the system buffer header table.

`semlock`	Array indexed by semaphore number. CPU bit mask indicates the CPU number who currently owns the lock.
`semowner`	Array indexed by semaphore number. Shows CPU number of lock owner and source file and line number where lock was set. For debugging purposes.
`semlinkf`	Array indexed by semaphore number. CPU bit mask of first (or only) CPU waiting for the lock.
`semlinkl`	Array indexed by semaphore number. CPU bit mask of last CPU in FIFO queue of CPUs waiting for the lock.
`cpulinkf`	Array indexed by CPU number. CPU bit mask acting as a forward pointer (index) to next CPU in FIFO queue.
`cpuhold`	Array indexed by CPU number. Source file name and line number where corresponding CPU referenced (and is waiting for) the lock. For debugging purposes.
`cpumemhold`	Array indexed by CPU number. For SEMLOCK type locks, the actual semaphore number the corresponding CPU is waiting on.

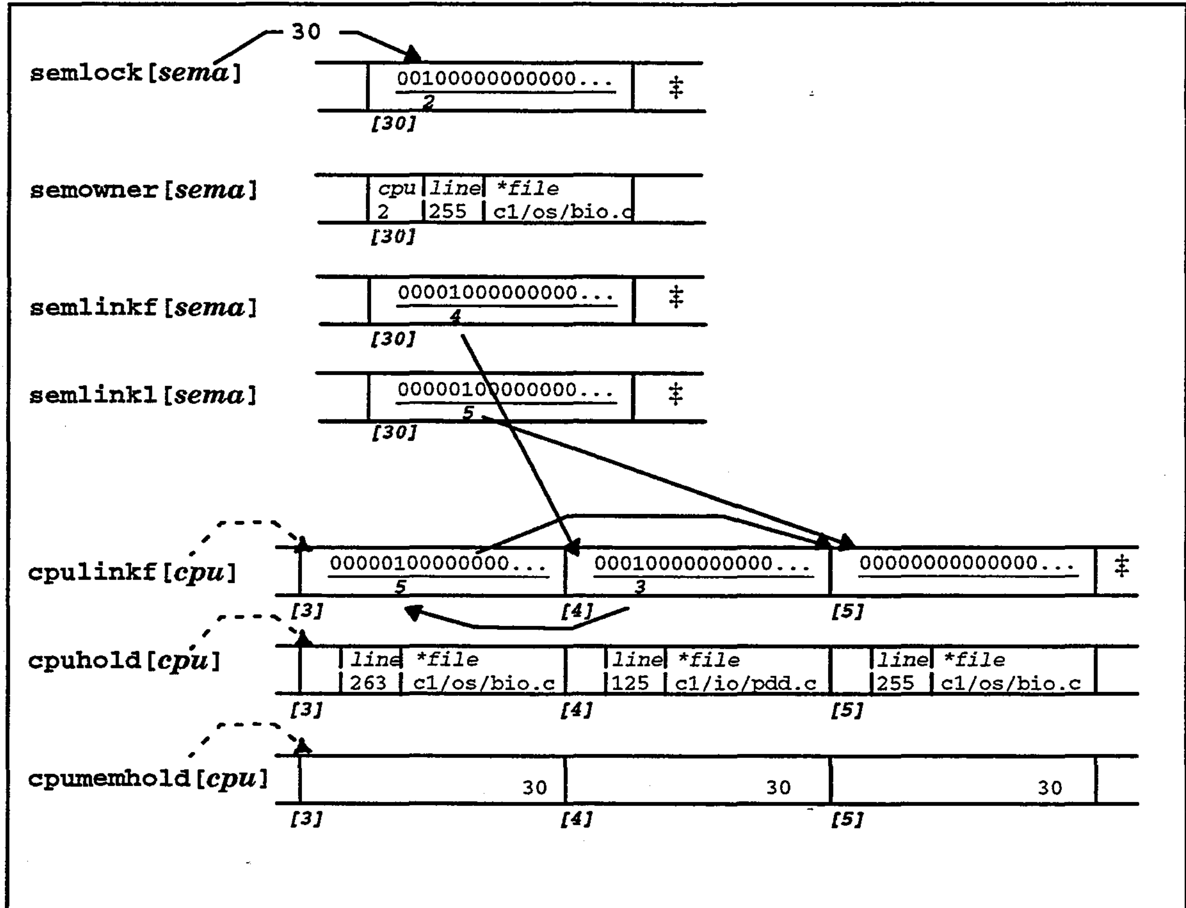
- The example shows a possible lock set referencing semaphore 30 (BUFLOCK).
  - CPU 2 is the lock owner. The lock was set at line 255 in source module `c1/os/bio.c`.
  - CPU 4 is the first (oldest) CPU waiting for the lock. It's reference was line 125 in `c1/os/pdd.c`.
  - CPUs 3 and 5 are waiting for lock 30 at lines 263 and 255, respectively, in `c1/os/bio.c`.
  - When CPU 2 executes macro SEMUNLOCK, CPU 4 will get the lock next, followed in turn by CPUs 3 and 5. Should any other CPU request the same lock, it will be queued following CPU 5.

SEMLOCK illustration

Note: The "sem" and "cpu" prefixed items are tables in `lowmem.c`. ♦

‡ *Fields marked with ‡ use a bit mask to indicate a set of CPUs, the position of the 1 bit from left to right indicates the CPU number.*

`c1/md/lowmem.c`



MEMLOCK macro

The MEMLOCK macro is used to test for and set a fine grained lock on a data area such as protecting the data in a single buffer header entry while the process is performing a system call in the kernel.

- MEMLOCK macro `include/sys/semmacros.h`
  #define MEMLOCK(sema, mem, lid)

sema	Semaphore number used to protect manipulation of this memory lock
mem	The address of semlck structure (include/sys/types.h) within the memory item being locked
lid	Lock id - unique number used as index to timing and lock rule tables

- A full set of macros defined in `include/sys/semmacros.h` reference MEMLOCK with proper sema, mem, and lid UNICOS kernel values. Example:
  `#define BUF_LOCK(semp)        MEMLOCK(BUF_ENT, semp, BUF_ENT_LID)`
- The diagram on the right shows key memory areas and illustrates how they are used in processing a lock on a system buffer header.

sem_lock	Semlck field within buffer table member. CPU bit mask indicates the CPU number who currently owns the lock.
sem_owner	Semlck field within buffer table member. Shows CPU number of lock owner and source file and line number where lock was set. For debugging purposes.
sem_linkf	Semlck field within buffer table member. CPU bit mask of first (or only) CPU waiting for the lock (this table entry).
sem_linkl	Semlck field within buffer table member. CPU bit mask of last CPU in FIFO queue of CPUs waiting for the lock.
cpulinkf	Array indexed by CPU number. CPU bit mask acting as a forward pointer (index) to next CPU in FIFO queue.
cpuhold	Array indexed by CPU number. Source file name and line number where corresponding CPU referenced (and is waiting for) the lock. For debugging purposes.
cpumemhold	Array indexed by CPU number. For MEMLOCK type locks, the address (table member) the corresponding CPU is waiting on.

- The example shows a possible lock set referencing buffer header pointed to by bp.
  - CPU 0 is the lock owner. The lock was set at line 35 in source module `c1/os/bio.c`.
  - CPU 10 is the first (oldest) CPU waiting for the lock. It's reference was line 623 in `c1/os/bio.c`.
  - CPUs 9 and 8 are waiting "in line" for buffer bp at line 125 in `c1/io/pdd.c` and line 263 in `c1/os/bio.c`.
  - Field cpumemhold array items for CPUs 8, 9, and 10 all contain the address of the semlck area within the currently locked buffer.
  - When CPU 0 executes BUFUNLOCK at the end of the buffer entry update, CPU 10 will get the lock and the table entry, and so on for CPUs 9 and 8.

MEMLOCK *illustration*

Note: The variable bp points to a buffer table entry buf which contains the structure semlck named b_sema. "sem" prefixed areas are within this buffer's b_sema field (reference bp->b_sema.sem_lock). "cpu" prefixed areas are tables in low memory.♦

‡ *Fields marked with ‡ use a bit mask to indicate a set of CPUs, the position of the 1 bit from left to right indicates the CPU number.*

bp

struct buf[NBUF]

struct semlck
b_sema

sem_lock 1000000000000... *0* ‡

sem_owner *cpu* *line* **file* 0 35 c1/os/bio.c

sem_linkf 00000000001000... *10* ‡

sem_linkl 00000000100000... *8* ‡

lowmem.c

cpulinkf[*cpu*] 00000000000000... 00000000100000... *8* 00000000010000... *9* ‡

[8] *[9]* *[10]*

cpuhold[*cpu*] *line* **file* 263 c1/os/bio.c *line* **file* 125 c1/io/pdd.c *line* **file* 623 c1/os/bio.c

[8] *[9]* *[10]*

cpumemhold[*cpu*] &bp->b_sema &bp->b_sema &bp->b_sema

[8] *[9]* *[10]*

ATOMIC lock macros

Atomic locks are used to protect a single data element for a single atomic operation, such as incrementing a count. These macros use memory lock logic but share a common set (table) of `semlck` data structures in low memory, using a hashing scheme to select a specific `semlck` table item.

- The atomic lock macros use the `SYS_LOCK` macro to indirectly reference the `MEMLOCK` macro to provide the lock control. `SYS_LOCK` is defined in `include/sys/semmacros.h`:
  `#define SYS_LOCK(semp, lid) MEMLOCK(SYS_ENT, semp, lid)`
- A full set of macros defined in `include/sys/semmacros.h` reference `SYS_LOCK` with proper `semp` and `lid` UNICOS kernel values for all common operations such as add, subtract, etc. The definition of a sample atomic lock `ATOMIC_ADD` is shown on the facing page.
- The atomic macros use macro `ATOMIC_HASH` to hash the user specified address, providing an index into the `atomic_locks` table in `lowmem.c`, where the lock is based.

  Note that several different atomic locks referencing different memory areas could all "share" this same lock - all CPUs would wait in turn across all of these locks. Atomic locks are very short duration locks so this is not a significant problem.
- The diagram on the right shows key memory areas and illustrates how they are used in processing an atomic add (increment) of the field `syswait.iowait`.

`sem_lock`	Field within `atomic_locks` array. CPU bit mask indicates the CPU number who currently owns the lock.
`sem_owner`	Field within `atomic_locks` array. Shows CPU number of lock owner and source file and line number where lock was set. For debugging purposes.
`sem_linkf`	Field within `atomic_locks` array. CPU bit mask of first (or only) CPU waiting for the lock (this table entry).
`sem_linkl`	Field within `atomic_locks` array. CPU bit mask of last CPU in FIFO queue of CPUs waiting for the lock.
`cpulinkf`	Array indexed by CPU number. CPU bit mask acting as a forward pointer (index) to next CPU in FIFO queue.
`cpuhold`	Array indexed by CPU number. Source file name and line number where corresponding CPU referenced (and is waiting for) the lock. For debugging purposes.
`cpumemhold`	Array indexed by CPU number. For `MEMLOCK` type locks, the address (table member) the corresponding CPU is waiting on.

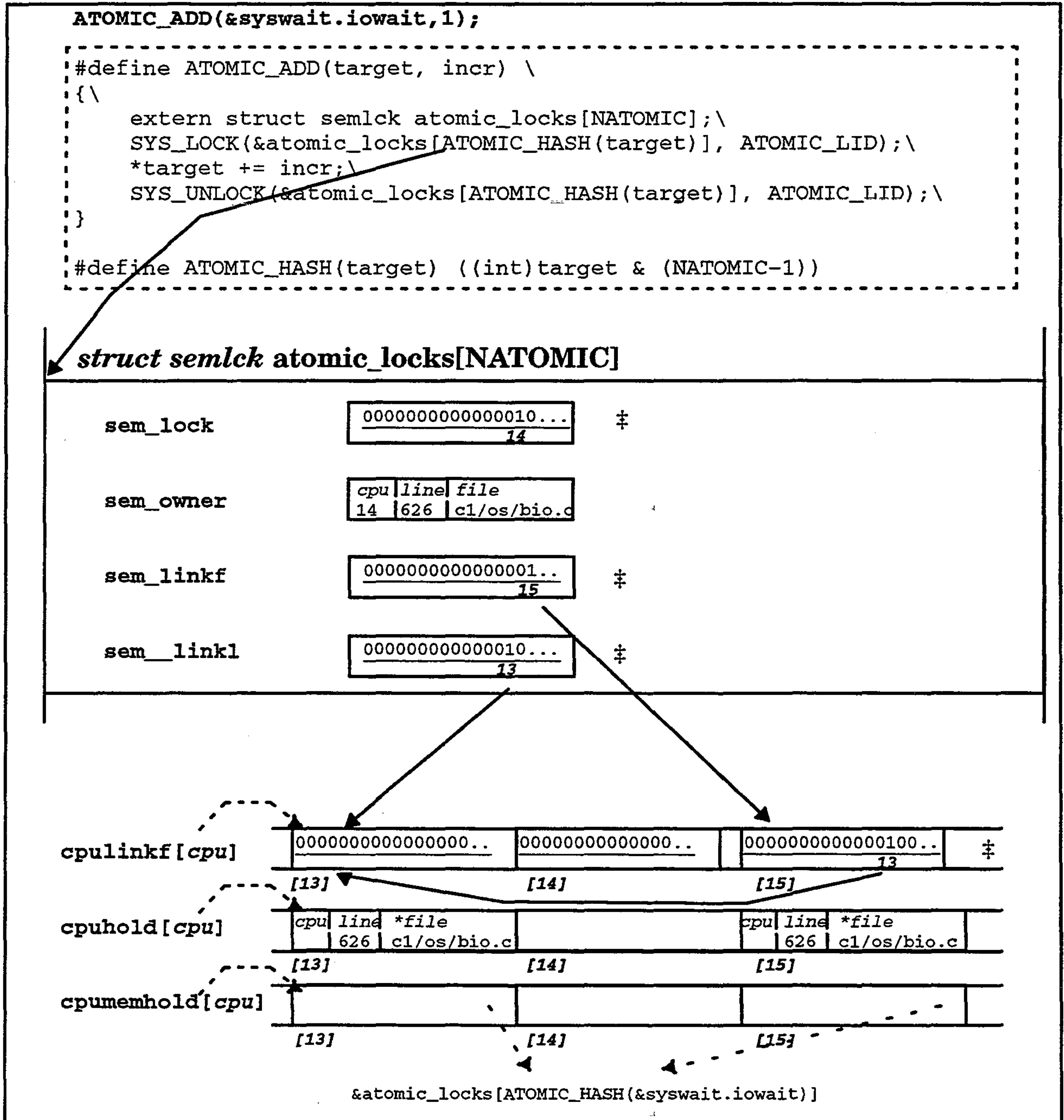
- The example shows a possible lock set for incrementing `syswait.iowait`.
  - CPU 14 is the lock owner. The lock was set at line 626 in source module `c1/os/bio.c`.
  - CPUs 15 and 13 are waiting for the same atomic lock. The address in `cpumemhold` for these CPUs would be the address of the `atomic_locks` table entry.
  - When the increment of the field is complete, macro `SYS_UNLOCK` releases the lock. CPU 15 will then get the lock, do its atomic operation, and proceed clearing the lock for CPU 13.

ATOMIC_ADD *illustration*

The atomic_locks and "cpu" arrays are in lowmem.c. The detail of the ATOMIC_ADD macro is shown in the inset.

‡ *Fields marked with ‡ use a bit mask to indicate a set of CPUs, the position of the 1 bit from left to right indicates the CPU number.*

lowmem.c



R_MEMLOCK and W_MEMLOCK lock macros

The memory read lock macro (R_MEMLOCK) and memory write lock macro (W_MEMLOCK) are variations of the MEMLOCK mechanism.

- A read lock allows any number of read accesses to the protected memory item, but do not allow a write (change of value) to the item as long as any read lock is set.
- A write lock is an exclusive use lock. Only the owner of the lock can reference the item. All other CPUs must wait for the lock whether reading or writing.
- Structure `rwsemlck` defined in `include/sys/types.h` contains two fields, `sem_rlock` and `sem_wlock` instead of the single field `sem_lock`, to provide for the read and write locks.
- R_MEMLOCK macro `include/sys/semmacros.h`
  `#define R_MEMLOCK(sema, mem, lid)`

`sema`	Semaphore number - used by this instance of the call
`mem`	The address of `rwsemlck` structure within the memory item being locked for reading
`lid`	Lock id - unique number used as index to timing and lock rule tables

  - R_MEMLOCK functions like MEMLOCK except that the CPU spin waits for the lock only if the item is locked for write (CPU flag(s) set in `sem_wlock`).
  - CPUs waiting for a read lock are queued on a FIFO queue of waiting CPUs.
  - Multiple CPUs can "own" a read lock on an item of data, `sem_rlock` shows a CPU bit for each occurrence. The last CPU executing R_MEMLOCK for a given data item shows as the lock owner in `sem_lock`.
- W_MEMLOCK macro `include/sys/semmacros.h`
  `#define W_MEMLOCK(sema, mem, lid)`

`sema`	Semaphore number - used by this instance of the call
`mem`	The address of `rwsemlck` structure within the memory item being locked for writing
`lid`	Lock id - unique number used as index to timing and lock rule tables

  - W_MEMLOCK functions like MEMLOCK except that the CPU spin waits for the lock if the item is locked for read or write (CPU flag(s) set in `sem_rlock` or `sem_wlock`).
  - CPUs waiting for a write lock are queued on a FIFO queue of waiting CPUs.
  - Only a single CPU can "own" a write lock at one time. However `sem_wlock` shows a CPU bit for each active or pending write lock. The first CPU executing W_MEMLOCK for a given data item shows as the lock owner in `sem_lock`.

- R_MEMUNLOCK macro include/sys/semmacros.h
  #define R_MEMUNLOCK(sema, mem, lid)

sema	Semaphore number - used by this instance of the call
mem	The address of rwsemlck structure within the memory item being locked for reading
lid	Lock id - unique number used as index to timing and lock rule tables

  – R_MEMUNLOCK clears the CPU's sem_rlock bit.

  – After clearing the last read lock the first CPU on the FIFO queue is allowed to proceed with the lock.

- W_MEMUNLOCK macro include/sys/semmacros.h
  #define R_MEMUNLOCK(sema, mem, lid)

sema	Semaphore number - used by this instance of the call
mem	The address of rwsemlck structure within the memory item being locked for reading
lid	Lock id - unique number used as index to timing and lock rule tables

  – W_MEMUNLOCK clears the CPU's sem_wlock bit.

  – After clearing the write lock the first CPU on the FIFO queue is allowed to proceed with the lock. Any additional CPUs waiting for read locks are allowed to proceed until the last CPU is processed, or a CPU waiting for a write lock is encountered.

- A full set of macros defined in include/sys/semmacros.h reference R_MEMLOCK and W_MEMLOCK with proper sema and lid UNICOS kernel values. Examples:

```
#define PROCTAB_READ_LOCK()   R_MEMLOCK(PLOCK,&proctab_lock,PLOCK_LID)
#define PROCTAB_READ_UNLOCK() R_MEMUNLOCK(PLOCK,&proctab_lock,PLOCK_LID)

#define PROCTAB_WRITE_LOCK() W_MEMLOCK(PLOCK,&proctab_lock,PLOCK_LID)
#define PROCTAB_WRITE_UNLOCK() W_MEMUNLOCK(PLOCK,&proctab_lock,PLOCK_LID)
```

- Multiple read and write locks are illustrated by an example on the following pages.

- The example shows the contents of a process table `proctab_lock` situation after the following series of read and write lock calls (without unlocks between). Only the lock and link fields in the `rwsemlck` structure are shown in the diagram.

CPU	**Macro Call**
2	PROCTAB_READ_LOCK()
5	PROCTAB_READ_LOCK()
3	PROCTAB_WRITE_LOCK()
7	PROCTAB_READ_LOCK()
0	PROCTAB_READ_LOCK()
4	PROCTAB_WRITE_LOCK()
1	PROCTAB_READ_LOCK()

  - CPUs 2 and 5 both own a read lock on the process table. They can access (but should NOT modify) its contents.
  - CPUs 3, 7, 0, 4, and 1 are all waiting on the read lock(s). Note - CPUs 3 and 4 are shown as having the write lock, but they are spin waiting. The write lock prevents CPUs 7 and 0 from getting the read lock until the CPU 3 write lock is processed.
  - The source file and line number (not shown specifically) in the `cpuhold` array would indicate where the kernel is waiting for locks for CPUs 0, 1, 3, 4, and 7.
  - The `cpumemhold` array indicates the memory address of the `rwsemlck` data item for CPUs waiting for the lock. It would be the address of `proctab_lock` in this example.
  - After both CPUs 2 and 5 unlock their read locks, CPU 3 will proceed with a write lock.
  - When CPU 3 unlocks its write lock CPUs 7 and 0 will both proceed with read locks.
  - After both CPUs 7 and 0 unlock their read locks CPU 4 will proceed with a write lock.
  - When CPU 4 unlocks its write lock CPU 1 will proceed with a read lock.
  - Any other locks set for this same area `proctab_lock` while the above is in progress will queue in turn after CPU 1's lock and be processed as described above.

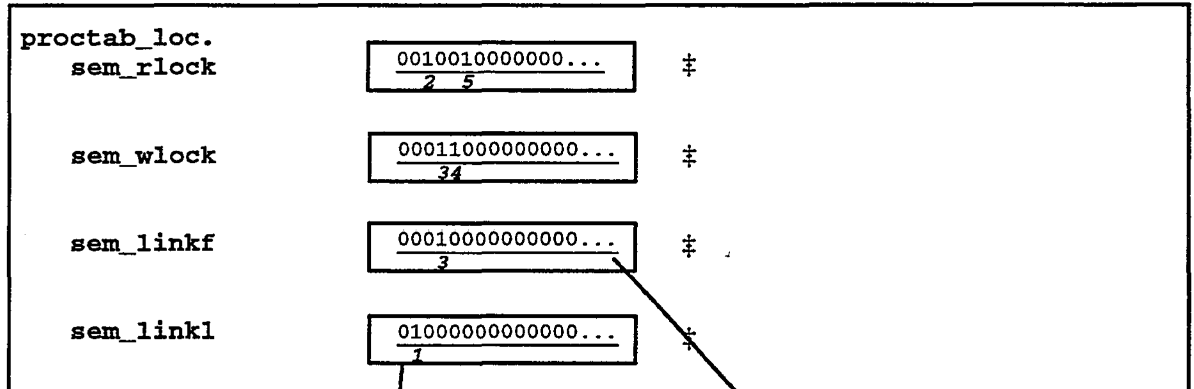
R_MEMLOCK and W_MEMLOCK illustration

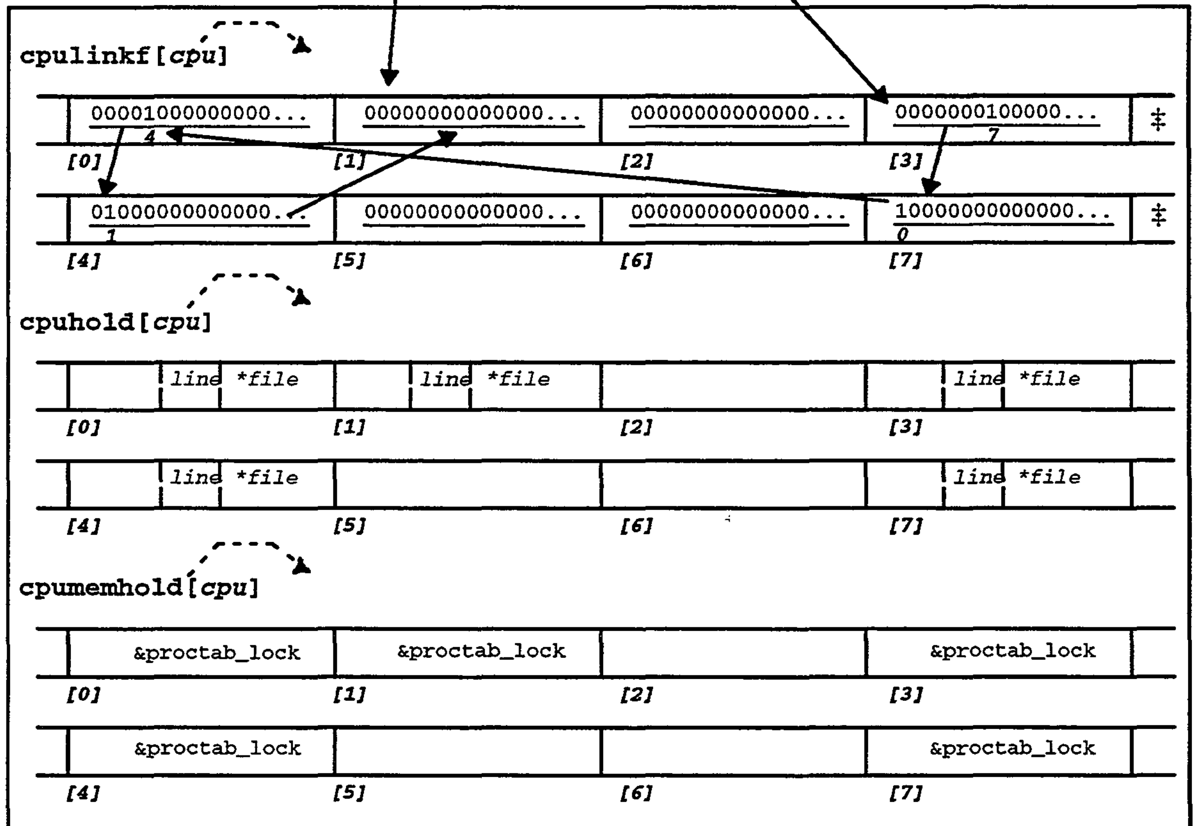
Field `proctab_lock` and the "cpu" arrays are in `lowmem.c`.

‡ *Fields marked with ‡ use a bit mask to indicate a set of CPUs, the position of the 1 bit from left to right indicates the CPU number.*

lowmem.c

struct rwsemlck proctab_lock;





Atomic sleep

The kernel sleep() function disconnects a CPU from a process, places that calling process on a sleep queue, selects a different process to run, and connects the CPU to that selected process. Since a process may sleep a very long time (get swapped out), extreme care must be exercised to insure locks are not held across sleeps.

To avoid holding a lock across a sleep, the pattern logic would be:

```
MEMLOCK(sema,mem,lid)
/* process locked data mem */
MEMUNLOCK(sema,mem,lid)
sleep()   /* CPU disconnects while waiting for some resource */
MEMLOCK(sema,mem,lid)
/* finish processing locked data mem */
MEMUNLOCK(sema,mem,lid)
```

Note: In the above that there is a window of CPU time between unlocking the protected area and the actual placing of the process on the sleep queue. Under certain conditions this can cause problems in the system. ♦

The buffer processing example on the upper right illustrates this problem.

Process "a", under lock, flags buffer as "busy" (1) while it is using it (2). Process "b" also wants buffer and locks it (3), finds it busy, and sets the "wanted" flag. While "b" is setting the flag "a" releases buffer (clear busy flag) but must wait on lock held by "b". Process "b" unlocks buffer (5) and calls sleep to disconnect and wait for buffer. Process "a" can then continue, locking the buffer and clearing the busy flag. "A" finishes by testing if any process has set the wanted flag and calls wakeup (6) to get any found started. If wakeup is completed in CPU 7 before sleep in CPU 1 places "b" on the sleep queue, the wakeup misses process "b" which may get "stuck" in the sleep queue.

Logic in sleep() prevents the above from happening.

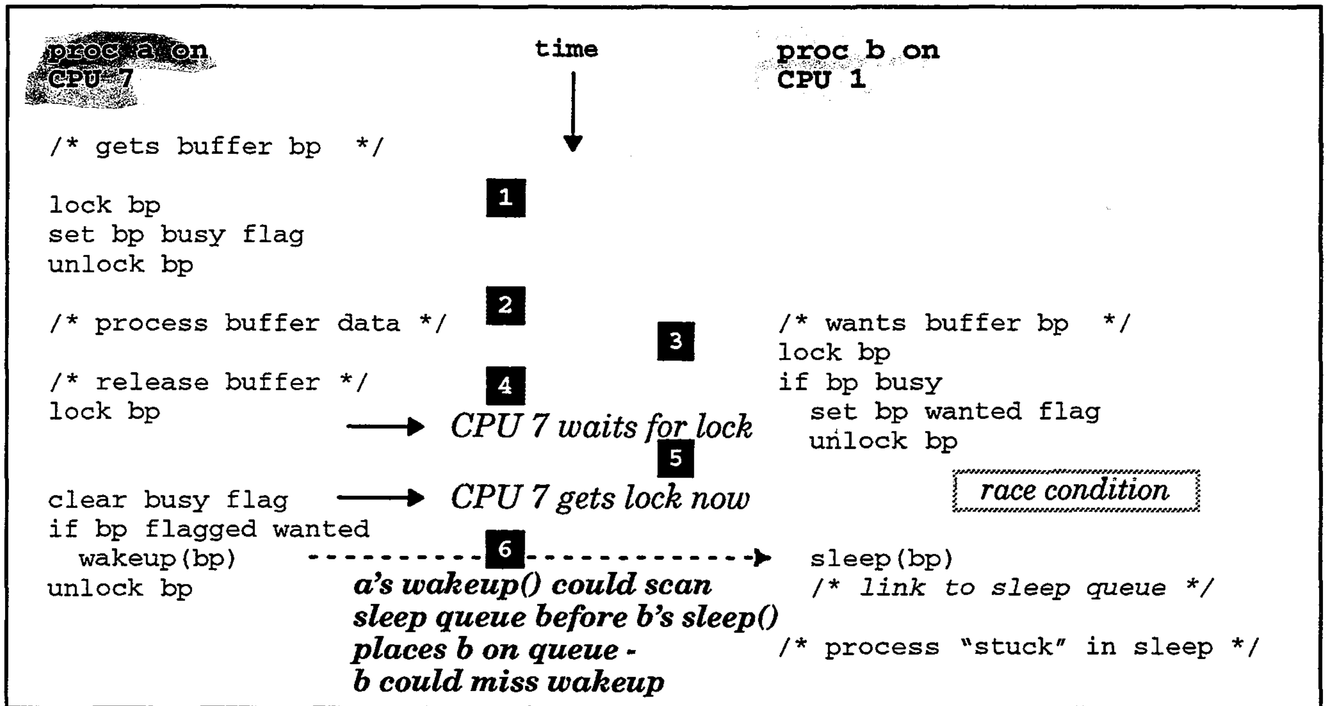
- Two fields in the process table entry indicate that sleep() should clear a lock:
  - p_slpsem   Semaphore number to clear.
  - p_slpmem   Address of semlck area in locked memory item
- Atomic sleep logic:

```
MEMLOCK(sema,mem,lid)
/* process memory mem */
p_slpsmem=sema
p_slpmem=mem
sleep()   /* sleep clears sema lock on mem before disconnecting CPU */
MEMLOCK(sema,mem,lid)  /* get lock after wakeup */
/* finish processing memory mem */
MEMUNLOCK(sema,mem,lid)
```

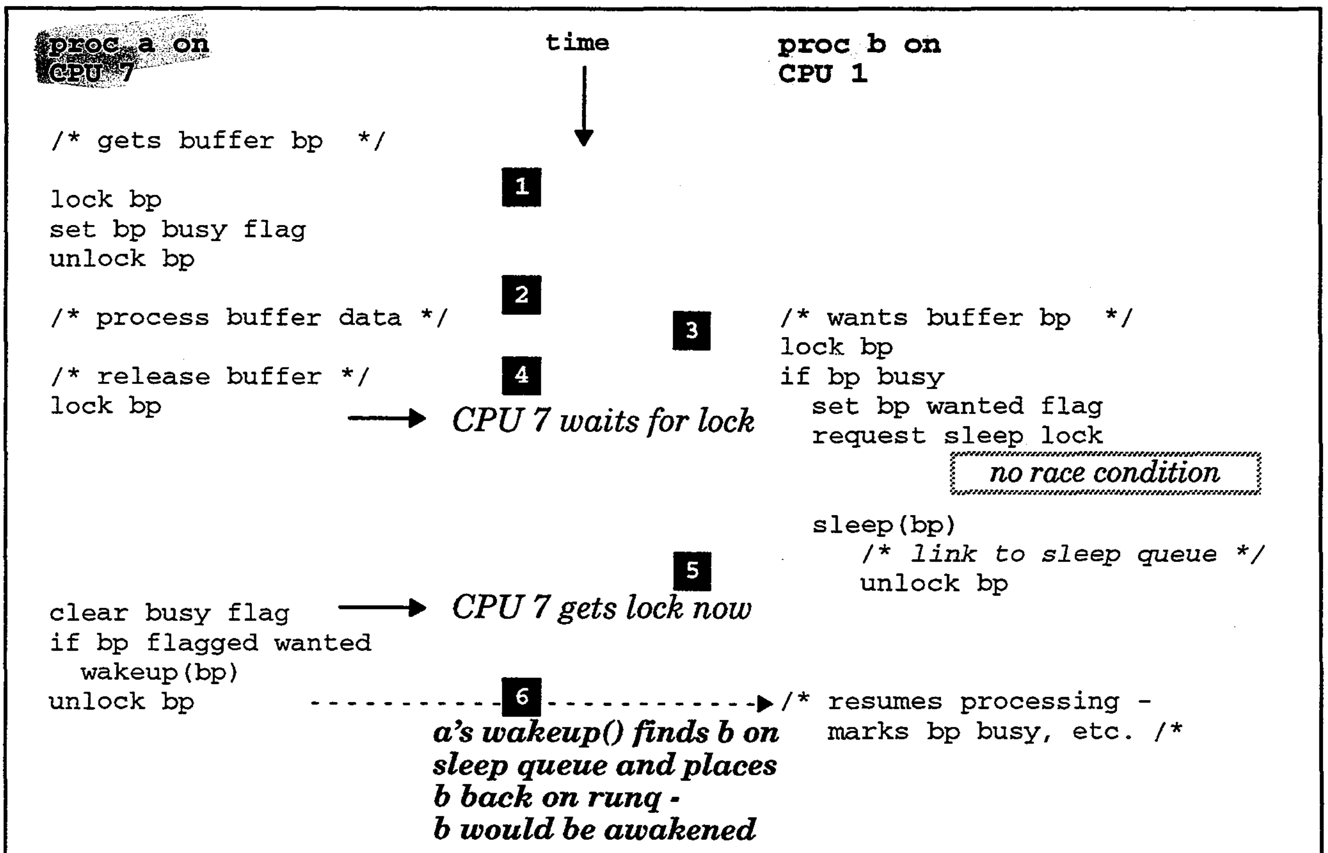
The setting of p_slpsem and p_slpmem causes sleep() to execute the MEMUNLOCK(sema,mem,lid) code for the semaphore p_slpsmem.

The example on the lower right illustrates the use of atomic sleep to correct the race condition problem. Process "b" requests that sleep unlock the buffer (3) causing "a" to wait (4) until the buffer lock is released (5). By the time "a" gets the lock "b" is on the sleep queue and will be awakened by "a" (6) so it can proceed.

Logic without atomic sleep



Logic with atomic sleep



Ownership macros

The following macros, defined in `include/sys/semmacros.h`, are available for the purpose of checking lock status and ownership:

- `ASSERT_LOCK` - PANIC if memory not set for this CPU
```
#define ASSERT_LOCK(mem)
   ASSERT(!multi_cpu || (mem)->sem_lock==((1<<63)>>cpu))
```
- `ASSERT_SEMLOCK` - PANIC if semaphore lock not set for the CPU
```
#define ASSERT_SEMLOCK(sema) \
   ASSERT(!multi_cpu || semlock[sema] == ((1<<63)>>cpu))
```
- `OWN_SEMLOCK` - TRUE if semaphore lock owned by this CPU
```
#define OWN_SEMLOCK(sema) (!multi_cpu || semlock[sema] == ((1<<63)>>cpu))
```
- `OWN_MEMLOCK` - TRUE if memory lock owned by this CPU
```
#define OWN_MEMLOCK(mem) (!multi_cpu || (mem)->sem_lock  == ((1<<63)>>cpu))
```
- `OWN_RWMEMLOCK` - TRUE if read or write memory lock set
```
#define OWN_RWMEMLOCK(mem) \
    (!multi_cpu || (((mem)->sem_rlock|(mem)->sem_wlock) & ((1<<63)>>cpu)))
```
- `OWN_MEMLOCK` - TRUE if write memory lock owned by this CPU
```
#define OWN_WMEMLOCK(mem) \
   (!multi_cpu || ((mem)->sem_wlock & ((1<<63)>>cpu)))
```
- `OWN_MEMLOCK` - TRUE if read memory lock owned by this CPU
```
#define OWN_RMEMLOCK(mem)  \
   (!multi_cpu || ((mem)->sem_rlock & ((1<<63)>>cpu)))
```

Lock hierarchy

To avoid deadlocks there must be an explicit hierarchy among locks that will be held concurrently.

- Each unique lock has a lock id assigned in `include/sys/semmacros.h`.
- A lock table `ltab` in `c1/md/lowmem.c` lists the locks in hierarchical order.
- A routine currently holding a lock can only set a lock lower than it's position in this hierarchy.
- The `SEMLOCKRULE` define builds a kernel with rule checking in place. This is not recommended for production systems due to CPU overhead. Lock rule violations are displayable with the `crash(8) leb` directive.
- Under certain conditions hierarchy violations are acceptable, such as when a locked data structure is destroyed or the lock is cleared in another process. The `LOCKRULE_CLEAR(lid)` macro corrects lock rule information in these cases.

Lock statistics

Lock statistics are gathered in array `lidstat[]` to show for each lock id `lid` the time spent waiting for the lock and the time a given lock is held.

Two site selectable options provide control over the collection of this data. Defining the `Nmakefile` symbol `SEMTIMING` generates additional code in the kernel to compute lock statistics, and the `lowmem.c` variable `semtswit` controls execution of this code.

The UNICOS kernel is released with `SEMTIMING` not set (commented out) and `semtswit` set to one (1).

- For each lock id's `(lid) lidstat` four statistics are computed:

`holdtime`	Time in (RTC) clocks CPUs have held for this `lid`. Computed by `SEMWAIT_END` and `MEMWAIT_END` macros if `semtswit` is non-zero.
`holdcount`	Count of times any CPU has held for this `lid`. Computed by `SEMWAIT_END` and `MEMWAIT_END` macros if `semtswit` is non-zero.
`locktime`	Time in (RTC) clocks CPUs have held `this lid`. Computed by `SEMLOCK_END`, `MEMLOCK_END`, `R_MEMLOCK_END`, and `W_MEMLOCK_END` macros if `SEMTIMING` enabled and `semtswit` is non-zero.
`lockcount`	Count of times any CPU has held this `lid`. Computed by `SEMLOCK_END`, `MEMLOCK_END`, `R_MEMLOCK_END`, and `W_MEMLOCK_END` macros if `SEMTIMING` enabled and `semtswit` is non-zero.

- Lock stastistics can be displayed with the `crash(8) mtstats` directive. Information is displayed for each lock id.

Note: During processing the `semtswit` value is loaded into global register `semtime` (_B(057)).♦

Lock debugging

Two `crash(8)` directives report information about kernel locks:

- `mtlock` displays the current state of a specific lock or all locks.
- `mthold` displays locks being held by by CPUs.

Additional `UTRACE` kernel trace line calls are provided for multi-threaded kernel debugging.

- The `crash(8) ut` directive displays kernel trace lines.
- Kernel Nmakefile symbol `SEMDEBUG` enables these trace lines.
- Tracing these lines is not recommended in production kernels. The system is released with `SEMDEBUG` not defined in the kernel `Nmakefile`.

Kernel register uses - *Kernel CPU register usage*

Reg(Oct)	Major B Register Usage	C Symbolic	CAL Symbolic
B000	Hardware return jump R saved p address		
B001	Current argument list pointer (stack)		
B002	Current stack frame pointer		B.%STKCBT
– – –			
B004	Jump switch pointer (interrupt handler)		B.SUBR
B005	Pointer to current process (A2 in master.s)		B.INDEX
B006	System call (function) number (user S0)		B.SYCALLN
– – –			
B050	macro scratch register	MACB2	B.MACB2
B051	macro scratch register	MACB1	B.MACB1
B052	macro scratch register	MACB0	B.MACB0
B053	kernel lock held usrioi	klockd	B.KLOCKD
B054	user XP address	xpuser	B.USERXP
B055	PWS entry address	pws	B.PWS
B056	non–zero if >1 CPU started	multi_cpu	B.MULTICPU
B057	semaphore timing switch	semtime	B.SEMTIME
B060	Used by Kernel flowtrace	KFTP	B.KFTP
B061	Pointer to current process pcomm	upc	B.UPC
B062	Pointer to current process proc	up	B.UP
B063	Pointer to current process ucomm	uc	B.UC
B064	Pointer to current process user	u	B.U
B065	CPU number "this" CPU	cpu	B.CPU
B066	Current Top of Stack Pointer		B%STKCTP
B067	Stack Limit Pointer		B%STKATP

Reg(Oct)	Major T Register Usage	C Symbolic	CAL Symbolic
T062	macro scratch register	MACT0	T.MACT0
T063	kernel profiling, macro	svs02	T.SVS02
T064	kernel profiling, macro	svs01	T.SVS01
T065	trace mask	tracem	T.TRACEM
T066	kernel profiling	frameofref	T.KPREF
T067	os lock start time	oslockrt	T.OSRT

Kernel cluster (1) register usage

Reg	SB (Shared B) Register Usage	C Symbolic	CAL Symbolic
– – –			
SB07	Count of number of parked CPU in usripi		ST.PARKCNT

Reg	ST (Shared T) Register Usage	C Symbolic	CAL Symbolic
ST00	pseudo– (channel) nterrupt mask – usrioi		ST.CHANF
ST01	channel lockout mask – usrioi		ST.CHANL
ST02	MEMLOCK macro wait mask		ST.MEMMASK
ST03	SEMLOCK macro wait mask		ST.SEMMASK
ST04	channel interrupt mask – real – usrioi		ST.CHANR
– – –			
ST06	I/O lockout flag		ST.IOFLAG
ST07	IPI mask – CPUs marked to go to usripi		ST.IPIMASK

Kernel cluster (1) semaphore register usage

Reg	SM (Semaphore) Register Usage –	C Symbolic	CAL Symbolic
	MEMLOCKs		
– – –			
SM02	Process common table entry lock	PCOMM_ENT	
SM03	Vnode lock	VNO_ENT	
SM04	DNLC cache of pathnames -> vnodes	DNLCLOCK	
SM05	File table entry lock	FILE_ENT	
SM06	Buffer structure lock	BUF_ENT	
SM07	Map structure lock	MAP_ENT	
SM08	Generic system table lock	SYS_ENT	
SM09	PCB element lock	PCB_ENT	
SM10	(NC1) Inode lock	INO_ENT	
SM11	Mbuf pool	MBLOCK	
SM12	SSD semaphore	SSDLOCK	
SM13	Filesystem lock	FSLOCK	
SM14	Run queue and switch stuff	RLOCK	
SM15	Global tables	GLOCK	
SM16	Kernel profile lock	KPLOCK	
SM17	Operating System lock bit	OSLOCK	
SM18	i/o register lock	IOLOCK	
SM19	Inode tables and routines	NLOCK	
SM20	Process tables and routines	PLOCK	
SM21	User tables and routines	ULOCK	
SM22	File tables and routines	FLOCK	
SM23	Sched tables and routines	SLOCK	
SM24	General shared register lock	HOLDLOCK	
	SEMLOCKs		
SM25	Request cpu park bit	PARK	
SM26	isema hold lock	ISEMA	
SM27	Cluster register mask lock	CGUARD	
SM28	Panic lock	PANLOCK	
SM29	History trace buffer lock	TLOCK	
SM30	Buffer cache lock	BUFLOCK	

Aliases for SYS_ENT*(SM08)*

SELECT_ENT
SOCK_ENT
SOCKQ_ENT
SOCKBUF_ENT
CHTAB_ENT
CCHTAB_ENT
NFSASYNQ_ENT
NFSCRED_ENT
SVDATA_ENT
NFSCKU_ENT
RNO_ENT

This page used for alignment

Bootstrapping the mainframe

Booting methods

There are two different methods available for booting the UNICOS kernel into the mainframe:

- Bootstrapping with the full kernel
- Bootstrapping with the compressed kernel

The compressed kernel option is the default method of booting the kernel into the mainframe released with UNICOS 8.0.

In the following description of bootstrapping the mainframe assume that the operator work station (OWS) has already been started.

Executing the `bootsys`(8) command on the OWS performs the following:

- Starts the `hbeat`(8) daemon to monitor for IOP halts and hangs, the `errlogd`(8) daemon to look for HISP errors, and the `smdemon`(8) daemon to monitor the OWS-E system for the system maintenance and remote testing environment (SMARTE)
- Runs IOP boot-time diagnostic tests (unless otherwise specified)
- Boots and configures each IOP
- Runs a set of mainframe diagnostic tests by executing the `mfinit`(8) program
- Boots the mainframe by executing the `mfstart`(8) program
- Executes the `zip`(8) command to provide you with the UNICOS console, unless you specify the -w ("without zip") option

The following pages describe the processing performed by the `mfstart`(8) command. See the `mfstart`(8) man page and the following man pages and Cray Research publications for more information:

- `configfile`(5) for information about the configuration file
- `owsepermfile`(5) for information about the default OWS-E permission file
- `bootsys`(8) for information about booting using the values from the UNICOS parameter file
- `rcpud`(8) for information about the remote CPU daemon
- `init`(8) for information about run levels in the *UNICOS Administrator Commands Reference Manual*, publication SR–2022
- *UNICOS System Administration*, publication SG–2113, for information about the UNICOS parameter file

Bootstrapping the mainframe with the full kernel

The Cray mainframe can be booted with a full kernel from the OWS. The full UNICOS binary (`unicos`), a startup parameter file (`param`), and an option boot root file system is provided as shown in the figure on the right.

The `mfstart`(8) command `-y`, `-p`, and `-f` parameters specify the unicos kernel file name, param file name and root file system name as shown. The boot root (or RAM root) file is normally only used for initial installation of the kernel or disaster recovery.

The OWS issues the command to the IOS cluster 0 to master clear the mainframe. The hardware master clear signal stops all mainframe CPUs and sets their XA registers to 0. The OWS provides the kernel binary, and parameter file which the IOS then writes into the mainframe's low memory.

The IOS is responsible for stripping off the loader header (exec structure) on the binary and for storing some values in the kernel's initial exchange package (location 0) that could not be known at compile time as follows:

A5: Number of the booting cluster (Model E)

A6: Number of the booting channel

S1: Length of the (optional) memory-resident root file system

S2 Length of the spare chip configuration file

S4: Size of the kernel binary

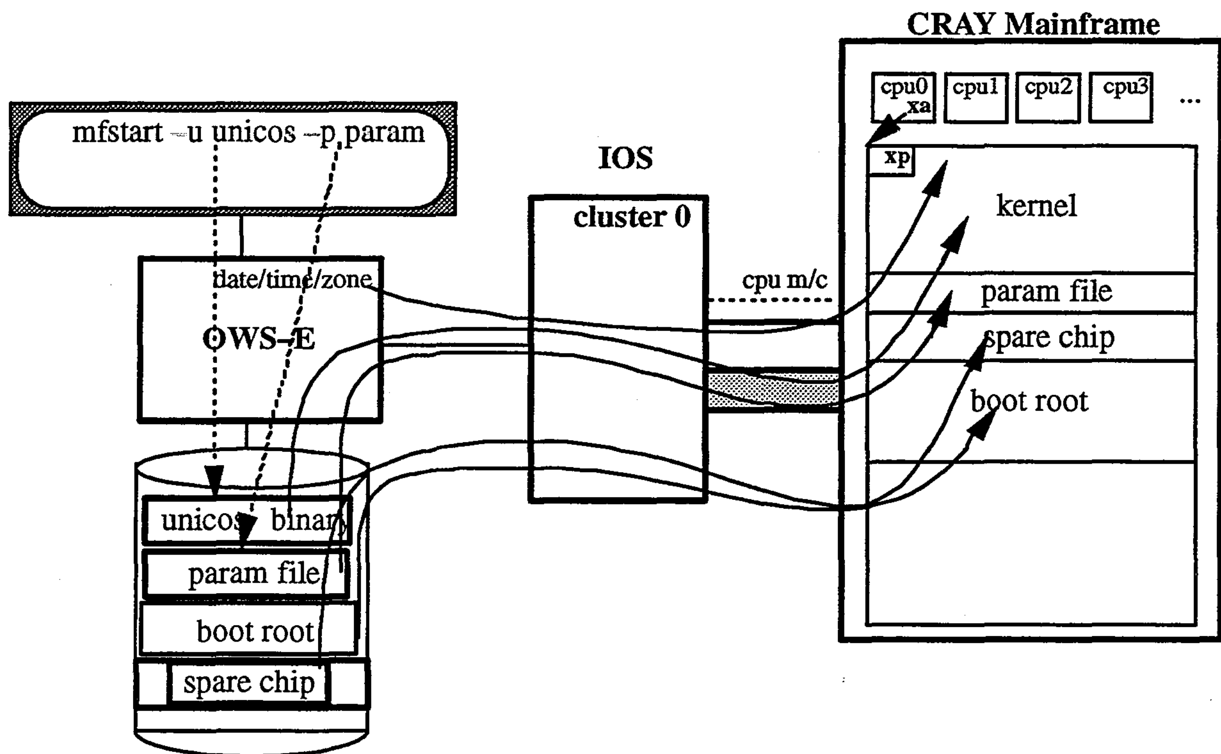
S7: Size of the kernel plus the startup parameter file

S6: Always 0 (Would be nonzero if `csim`/`ncsim` was executing the kernel)

S5: This register is left 0, indicating that this boot was done by the IOS and therefore the kernel must perform the initial handshaking that the IOS expects.

The IOS releases the master clear, causing an interrupt in CPU 0. (The other CPUs stay in the master-cleared state until they receive an interprocessor interrupt.) CPU 0 begins executing at `mfstart` in the kernel.

Booting the Mainframe with the Full Kernel



Kernel structures at deadstart

CPU 0 exchanges the kernel's initial xp (at address 0) into its registers and begins to execute kernel code at the address compiled into the P register of that xp ("mfstart" – see c1/md/lowmem.c).

Central memory as CPU0 enters the kernel is shown in the figure on the right:

S5 = 0 if loaded from IOS

If a kernel "wakes up" with a zero value in S5, it knows that it has been loaded from the IOS (that is, not through a mainframe bootstrap of itself).

S6 = 0 if not loaded by csim

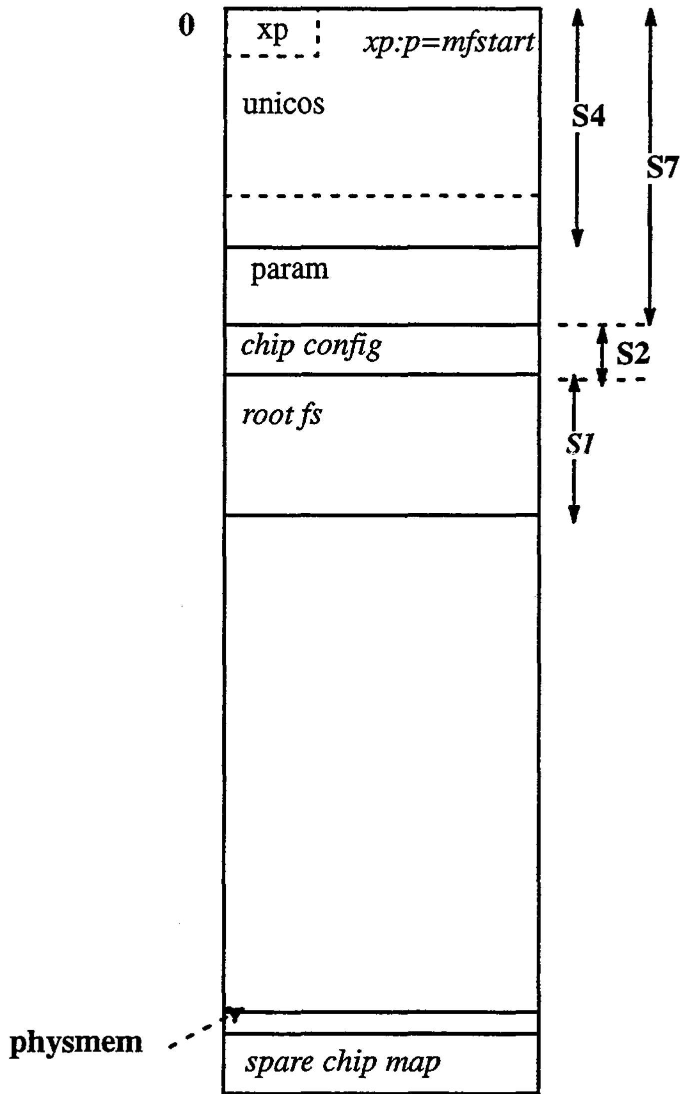
If the kernel "wakes up" with a zero value in S6, it knows that it is being executed directly by real hardware, and that the execution of its code is not being done by a Cray simulator (csim).

The value of S6 is saved as a csim flag in location 0176.

Certain situations test this csim flag so that the code runs faster under csim as follows:

- ddinit() routine won't attempt flaw initialization
- The T packet (real terminal – Model D) driver putchar() won't delay
- panic() won't attempt I/O to the IOS
- The semsleep macro will report the CPU idle waiting for a semaphore

Kernel structures at deadstart



Bootstrapping the mainframe with a compressed kernel

Beginning with release 7.0, the kernel `Nmakefile` builds and executes a utility named `kcompress` to compress the kernel. Source is in `uts/cmd/c1/kcompress`. The figure on the right shows a diagram of a compressed boot kernel.

Compression of the kernel saves about 70% on the size of the deadstart binary. By compressing the kernel binary, OWS disk space is saved and the boot time is shortened because there is less I/O activity required to read the kernel binary.

The `kcompress` utility writes a decompression routine into low memory, saves the kernel initial exchange package in low memory (at location `045` hexadecimal) and replaces it with an initial exchange package to execute the decompression routine. The first 128 words of the kernel are not compressed. Only the kernel binary is compressed, not the symbol table at the end of it (needed by `/etc/crash` utility).

At deadstart, CPU0 exchanges to the decompression routine which moves the parameter file and symbol table up, expands the kernel binary, copies the kernel initial exchange package back down to 0 and jumps to the kernel's entry point (`mfstart()`).

The `/unicos` file is a copy of the decompressed kernel. Decompression is a function of the mainframe and not the IOS, so this works equally well with IOS model B, C, D, or E.

The DECOMP() routine maintains the register values provided by the IOS in the initial exchange package. It also bumps up S4 and S7 to reflect the size of the kernel and parameter file after decompression.

DECOMP() moves the memory-resident root file system also.

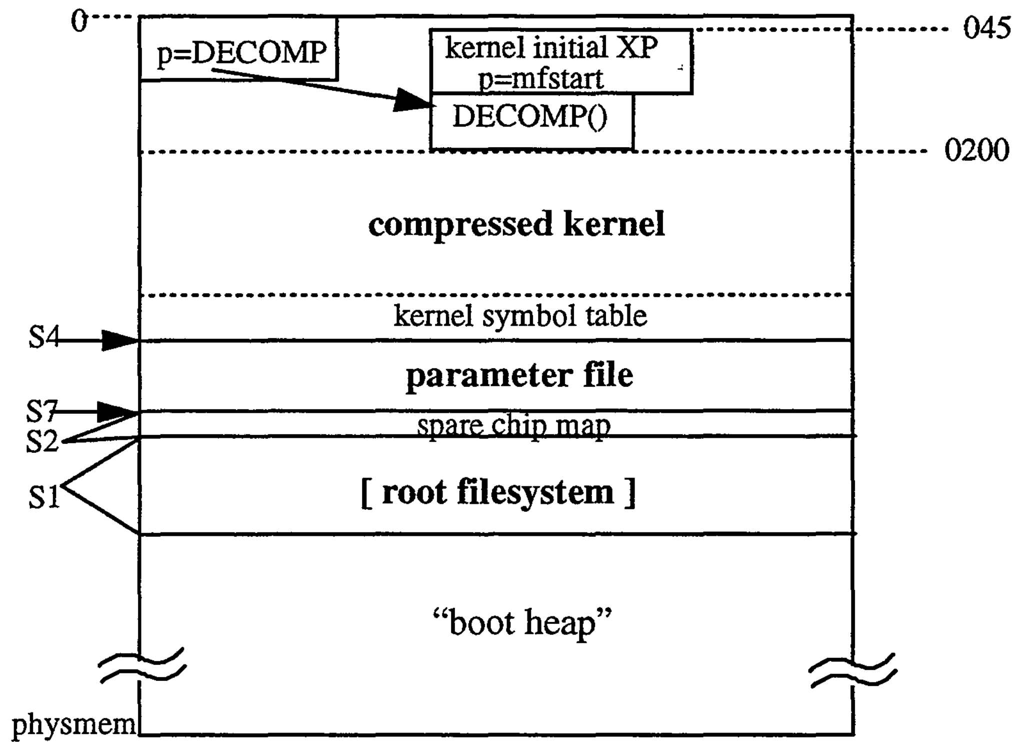
The compression of the kernel is reflected in the `exec` structure attached to the beginning of the binary. Use the `size`(1) command to show the text, data and bss of a kernel binary as follows:

uncompressed kernel: 844021 + 0 + 0 = 844021 words

compressed kernel: 276251 + 0 + 567770 = 844021 words

The bss size represents the space saved by compression.

Compressed Kernel Boot



This page used for alignment

UNICOS kernel startup

Startup overview

The following section describes kernel logic flow starting from CPU 0's entry into the kernel at `mfstart` through the initial entry of `init(8)` (`/etc/init`) executing in single user mode.

- Stack initialization
- Hardware initialization
  - CPU 0
  - CPUs 1-n
  - Clusters, etc.
- Startup parameter file processing
- Starting UNICOS
  - System process creation
    - `init`
    - idles
    - `esd_pulse`
    - `utility`
  - Mounting the root file system
  - Entering `sched()`
  - Entering `init(8)`

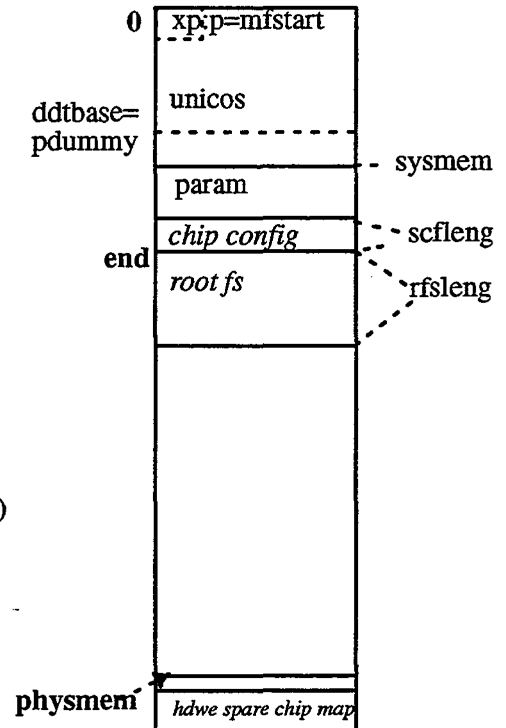
mfstart / mfinit logic

Key elements of kernel routines mfstart and mfinit follow. Memory contents are shown in the diagrams on the right.

- Memory mapping address contained in the initial exchange package, and now in A and S registers are saved in memory variables shown.
- Macro SETGBT loads kernel global registers as shown in the diagrams.
- Address ddtbase is set to the future base of the driver tables. (They will be located there after they are built.
- The stack area for process 0 is initialized by
  - Aligning the zerostk address to a *click* boundary within process zero's ucomm and user area reserved for it in low memory.
  - Setting stack control registers B002, B066, and B067.
  - This stack will support function call/return logic in the c language routines to follow.
- mfstart jumps to mfinit (no stack frame is created yet).
- After setting "in init" flag init, routine:
  - pbinit sets initial ASCII names and pointer in the panic buffer.
  - uninit sets initial ASCII names and pointer in the kernel trace buffer.
  - machinfoinit initializes the machine info. table values to zero.
- The deadstarting CPU's XA is set to point to it's "unix" exchange package area in its processor working storage table area. After this (unix XA is changed) any interrupt would cause the CPU to go to immtrap causing a system panic.
- The CPU is switched into hardware cluster 1, the system cluster. Cluster 1 is zeroed.
- Memory is physically scanned to determine highest address, saved in discmem. Memory from sysmem+rfsleng thru discmem zeroed.
- IOS model B, C, and D only: miopinit reads data/time from MIOP, swaps I and J initialization packets, and sets up miop table.
- If (C90) spare chip map (scfleng!=0) copy it to sparechip in low memory.
- Detail about csl() processing, startup file creation, and table relocation is shown on the next 4 pages.
  - If IOS model E calls csl() to process param file, configuration specification language (CSL) directives in the startup parameter file are used to create "driver tables" for the kernel.
  - If IOS model B, C, or D calls pscan() to process the startup parameter file (detail not shown).
  - Memory copies of startup files are created and saved in high memory (written to root after it is mounted).
  - The driver tables are relocated down (to ddtbase) reclaiming "user" memory.

Bold is a routine name!!

mfstart / mfinit logic

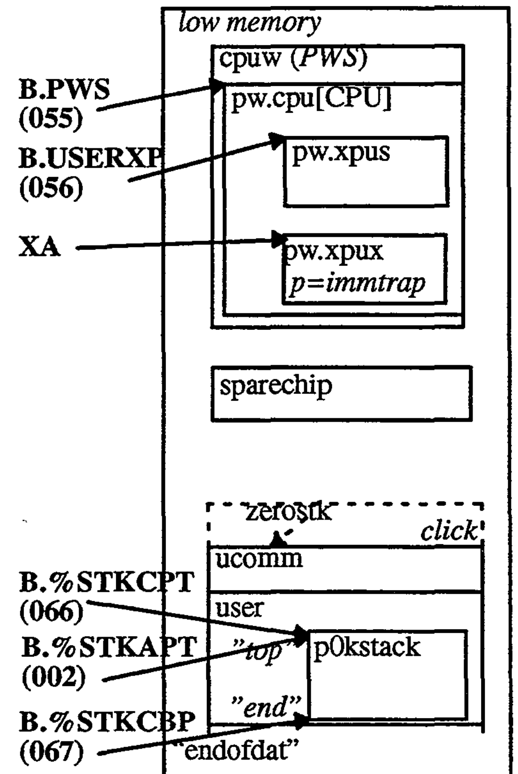


mfstart *md/mfinits.s*

owslink=A5 (*boot cluster*), owschan=A6 (*boot channel*)
rfsleng=S1 (*length of root fs*)
scfleng=S2 (*chip config*), sysmem=S4 (*len. UNICOS binary*)
bootload=S5, csim=S6, end=S7 (*unicos+spare+param*)
owsplvl=A7(*bit 40-47: IOS protocol level*)
irunlvl=A7(*bit 48-55: init run level*)
SETGBT (*set global B and T registers macro*)
- B.PWS (055)=PWS (*pws cpu entry "this cpu"*)
- B.USERXP (056)=pw.xpus (*user XP area for this CPU*)
- B.CPU (065)=PN (*CPU*)
- B.KFTP (060)=0 (*kernel flowtrace pointer*)
- B.SEMTIME (057)=semtmswit (*semaphore timing*),
- B.TRACEM (065)=tracemask (*semaphore trace*)
- B.MULTICPU(056)=multicpu (*non-zero* for multi-thread kernel)

ddtbase= &pdummy
(if !S5) clear lowspeed channels
C90: disable interrupts all channels
prepare the stack pointers for entry to C code:
- align proc[0]'s "swap image" zerostk on a click boundary
- B.%STKCTP(066)=p0stack (*base of stack*)
- B.%STKCBP(067)=p0stack+KSTACKL (*end of stack*)
- B.%STKATP(002)=0 (*top of current stack*)

XA=pw.xpux (*"UNIX" exchange package for this CPU*)
(*Jump to mfinit*)



mfinit *cl/md/mfinit.c*

ininit++
- **pbinit** *cl/md/machinfo.c* (*initialize the panic buffer*)
- **utinit** *cl/sys/sysmacros.h* (*initialize the trace buffer*)
- **machinfoinit** *cl/sys/sysmacros.h* (*setup machinfo table*)

initialize XP at pw.xpux (*would call immtrap*)
XA=pw.xpux
CLN=1 (*switch to cluster 1 - system cluster*)
clear CLN 1 registers
`discmem`=("scanned for" end of memory)
clear `sysmem+rfsleng` thru `discmem` to zero
(IOS B/C/D only): **miopinit** (*init IOS IOP*)
adjust `physmem` down to compensate for ila/dla truncation
- *scfleng != 0*
  - copy "chip config" to sparechip (*in lowmem*)
- *IOS E*

pscan() **csl()** (*process parameter file*)
create "files" at end of memory
relocate driver tables next to kernel
(*continued on next page*)

csl processing

The csl() function builds system tables following the kernel area in memory from the information in the ASCII parameter file as follows:

- z_ios_parp points to sysmem + scfleng
- z_parser() to processes parameter information storing it in a scratch memory area (heap).
- relfac = cf_data – ddtbase
  Pointers to and within table created by csl() are relocated by relfac before returning.
- csl() itself and functions it calls allocate and initialize the data/table items as shown in the table below.

Function	**Description**	**Data Item / Table**	**Description** **ASCII label: contents**
csl_initIOS()	IOS initialization	miop	*miop: MUX* IOP table
		iostab	*iosdtbl: IOS* device table
		eiopack	*eiopkts*: E packet table
		epackend	End of E packet table
csl_initmf()	Mainframe initialization	cpquan	Number of CPUs, must be <= configured
		mi_maxclus	Number of clusters, must be <= configured
		physmem	Size of main memory, must be <= configured
		halfmem	Selected half of C90 memory
		chant	Initialize only (alloc in lowmem)
csl_initunicos()	UNICOS initialization	nbuf	Number of system cache buffers
		v.nbuf	Number of ldcache headers
		v_ldchcore	LDCHCORE
		LDDEVCT	Maximum number of ldd devices
		slice_prof [ldd_major]	ldd device slice table
		ddmaps	*ddmaps*: disk device maps table
		nldmap	Number of ldmap items
		slice_prof [mdd_major]	mdd mirrored device slice table
		mdd_tab	*mdd_tab*: mirrored device table
		hdd_tab	*hdd_tab*: HIPPI disk device table
		HDDEVCT	Maximum number of hdd devices
		slice_prof [hdd_major]	hdd HIPPI disk device slice table
		pdd_tab	*pdd_tab*: physical (dd) device table

Function	Description	Data Item / Table	Description ASCII label: contents
		DDEVCT	Maximum number of hdd devices
		slice_prof [rdd_major]	rdd ram device slice table
		slice_prof [sdd_major]	sdd striped device slice table
		sdd_tab	*sdd_tab*: striped device table
		slice_prof [ssdd_major]	ssdd SSD device slice table
		v_tp_bufz	Max. buffered (block) size for tape
		v_tp_conf_up	Maximum tapes configured up
		v_tp_max_dev	TAPE_MAC_DEV???
csl_initfs()	File system initialization	root_conf	ldd config info (eslice/dd_tab) for the root file system device
		swap_conf	ldd config info (eslice/dd_tab) for the swap file system device
		swapunits	swap dev size(blocks) / swp_wght (swp_wght = 16)
csl_initram()	Ram disk initialization	ramsize	Ram disk size converted to words
csl_initSSD()	SSD initialization	ssd_count	Number of SSD devices
		vhspconf	*vhispcf*: VHISP channel configuration table
		ssdconf	*ssdconf*: SSD configuration table
		ssdd_tab	*ssddtab*: SSD device table
		SSDDMAX	maximum SSD devices
		sdsbits	*sdsbits*: SDS bit map table
csl_inithi()	HIPPI driver initial.	himaxdevs	Channel table size
		himaxpaths	Number of paths
		nhippi	Number HIPPI channels
		hidev	HIPPI device table
csl_initnp()	LOWSP comm driver initial.	np_vars	Initialize only
		np_devs	n packet driver control table
csl_initmb()	TCP only: Mbuf param initialization	v_tcp_nmbspace	Space for mbufs; allocated in coremap by minit() later in umain()
csl_initnfs()	NFS only: NFS params	v_nfs_num_rnodes	Number of rnodes
		v_nfs_static_clients	Num. of static client handles (nfs)
		v_nfs_temp_clients	Num. of temp. client handles (nfs)
		v_cnfs_static_clients	Num. of static client handles (cnfs)

Function	**Description**	**Data Item / Table**	**Description** **ASCII label: contents**
		`v_cnfs_temp_clients`	Number of temp. client handles (cnfs)
		`v_nfs_maxdata`	Max. user data read/written
		`v_nfs_wcredmax`	Max. number of credential structures
		`v_nfs_maxdupreqs`	Max. duplicate request cache
		`v_nfs_duptimeout`	Duplicate replay timeout
		`v_nfs_printinter`	Time out error redisplay interval
`csl_init_fddi()`	ELS only: EL FDDI driver initialization	`nfddi`	Max. number of FDDI devices
		`fddi_devs`	FDDI device table
`csl_init_en()`	ELS only: EL Ethernet driver initialization	`en_devs`	Ethernet device table
`csl_initfd()`	Not ELS: FDDI driver initialization	`nfddi`	Max. number of FDDI devices
		`fd_devs`	FDDI device table
`csl()`	Control statement processor	`cf_text`	Beginning of parameter file area in memory
		`cf_text_bl`	Length of parameter file area
		`cf_data`	Base of system table area
		`swapbits`	*swapbits*: swap device allocation bit map
		`mcachebits`	*mcachebit*: memory cache (systen buffer) allocation bit map
		`nbuf, nhbuf`	Round up to next power of 2
		`cf_data_wl`	End of system table area

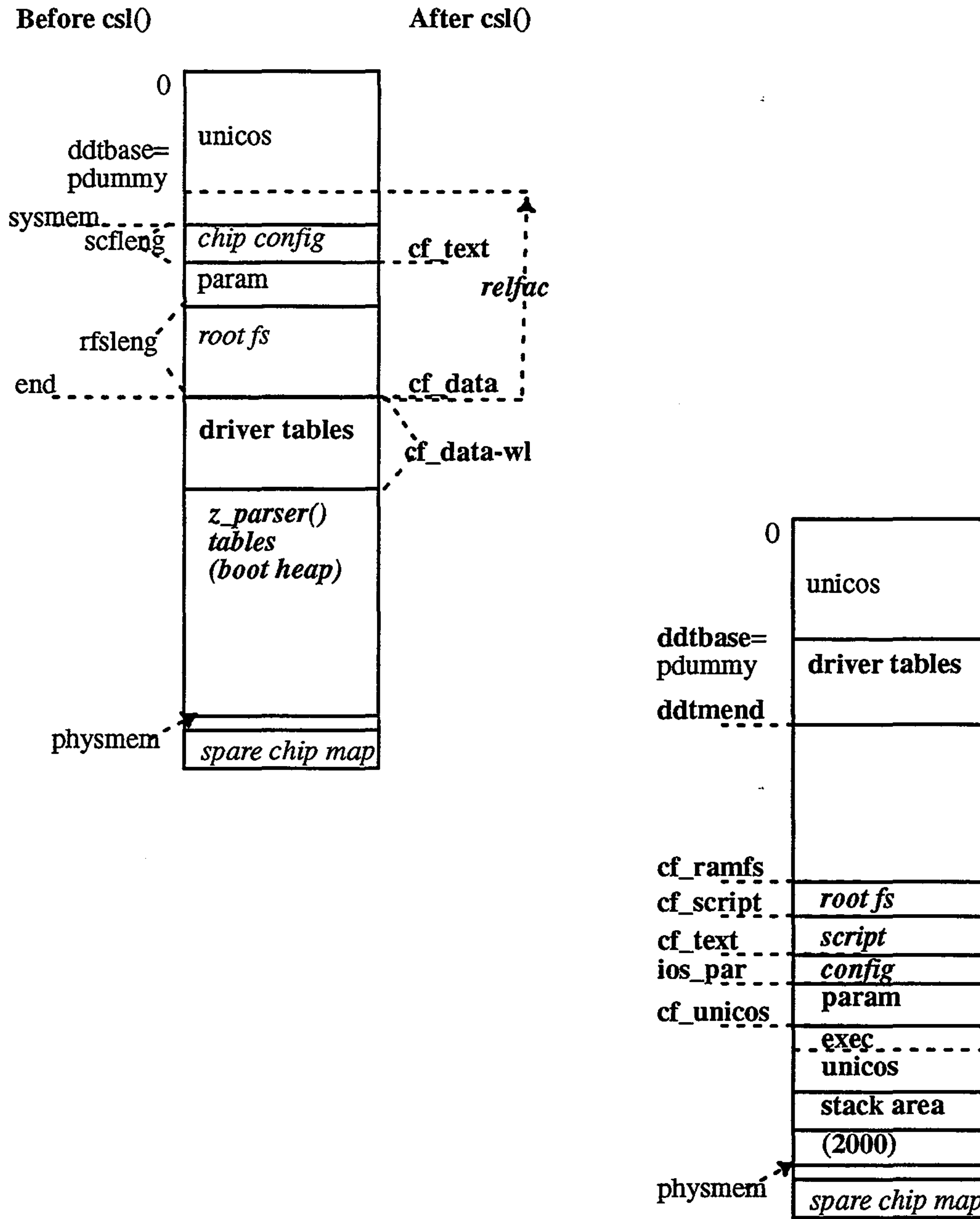
This page used for alignment

startup file / table relocation

The diagrams on the right illustrate how memory information is located and relocated during startup processing.

- `csl()` processing
  - `cf_text` points initially to the base of the parameter file (`cf_text_wl` is its length.
  - `z_parser` processed parameter information in the *boot heap*.
  - Various "z_" functions (preceding page) build driver tables located at `cf_data` (`cf_data_wl` is its length).
- Items are saved in high memory while the rest of startup continues.
  - `/unicos` binary - used by system commands and crash(8).
    - ▲ An initial kernel stack area (plus 2000 words) is reserved just below `physmem`.
    - ▲ A standard a.out header (`struct exec include/sys/aoutdata.h`) is built at `cf_unicos`.
    - ▲ The kernel binary image (including symbol tables) is copied after the header.
    - ▲ `cf_unicos_wl` is length of whole area.
  - IOS parameter file (`/IOS_param` on IOS B/C/D systems)
    - ▲ Copy `param` file contents to `iospar` (now relocated to high memory)
    - ▲ `ios_parwl` is its length.
  - IOS configuration file from `pscan()` (`/CONFIGURATION` on IOS B/C/D systems)
    - ▲ Copy configuration file contents to `cf_text` (now relocated to high memory)
    - ▲ `cf_text_wl` is its length.
  - IOS script from `pscan()` (`/etc/setdev` on IOS model B, C, and D systems)
    - ▲ Copy script file contents to `cf_script` (now relocated to high memory)
    - ▲ `cf_script_wl` is its length.
  - Optional boot root file system.
    - ▲ If present (`rfsleng!=0`) and its image is overlapped by the "future" relocated driver tables, copy it to `cf_ramfs`.
    - ▲ `cf_ramfs_wl` is its length.
  - Driver tables (at `cf_data`) are copied down to `ddtbase` (or `pdummy`). Note `csl()` or `pscan()` and their related "z_" functions and other startup file data is reclaimed (overlaid) by this action. `ddtmend` is the end of the driver tables (and the start of other system table information.

startup file / table relocation



mfinit()logic (continued)

- `emiopinit` initializes IOS Model E tables, gets date and time from IOS, and initializes packet queuing tables.
- Initializes LOWSP and VHISP channels.
- `icpu` initializes CPUs 1-n
  - Clears IPI interrupts.
  - Builds Xp for each new deadstarted CPU "n" (p.aaddr = `park`).
  - Sets OSLOCK (single thread kernel).
  - Initializes PWS for CPU 0.
  - For each CPU "ncpu" 1 through n:
    - ◊ Initializes CPU[ncpu[PWS area.
    - ◊ Create new deadstarting XP (p=park).
    - ◊ Sends IPI to CPU[ncpu]
    - ◊ Spins until CPU "ncpu" write "INIT" in PWS
    - ◊ Sets machine information table data.
  - CPU ncpu starting in `park`:
    - ◊ Sets its global register to reference its idle process.
    - ◊ Initializes its XA, clock, interrupt flags.
    - ◊ Save "INIT" in PWS.
    - ◊ ===========
Spin until OSLOCK cleared.
===========
    - ◊ Resets stack to base frame.
    - ◊ Call `mcpu` to complete initialization.
    - ◊ Jump to `master` (enter kernel mainline loop)
- Sets CPU 0 programmable clock for 1 second (from now) interrupt.
- Sets system date and time fields.
- Build error exchange package "trap" at location 0 (PANIC zeroXP).
- Trace going to `gounix`.
- Calls `gounix`:
  - Initializes stack (eliminates gounix frame).
  - Calls `umain`:
    - ◊ Finishes initialization (following pages for detail).

Note: `gounix` returns logically once for process 0 and once for each idle process (reference from `park` above).♦

– *Continued on following pages after return from* `umain`.

This is what other CPU's start with - NOTICE

mfinit()logic (continued)

IOS E only: **emiopinit** *c1/md/einit.c*
- Send time and date request to the OWS.
- **eiosetup** *c1/md/einit.c*
  - initialize the free packet list
  - run through all the clusters/IOPs and
- initialize the appropriate channels

IOS B/C/D only:
- add packets to the packet free list

ssdinit *c1/md/init.c* initialize SSD channels
mppinit *c1/io/mppstub.c* (stub)
icpu *c1/md/icpu.c*
- **CIPI** *c1/sys/asm.h* clear IPI interrupts
- **clrsrs** *c1/md/icpu.c* clear status registers
- IB/DB=0 IL/DL=physmem
- **setperf** *c1/md/perfmon.s* initialize HPM regs
- clear all cluster regs
- **SEMLOCK** : set OSLOCK
- initialize PWS area for CPU0 – show CPU 0 started
- for ncpu = 1 to cpquan
  - initialize CPU[ncpu] PWS entry
  - create new XP in unixxp (p=park)
  - setxp c1/md/icpu.c finish XP – copy it to 0
  - **SIPI** *c1/sys/asm.h* sent interrupt to CPU ncpu
  - spin wait until PW[ncpu] sets "init" or time out
    - time out – mark CPU ncpu down
    - INIT – mark CPU ncpu up
  - **SetMachineInfo** *c1/sys/machinfo.h*
    - build default machinfo data

set p_clock = HZ (1 second)
rtcinit *c1/md/mfinit.c*
- tz=time zone
- timbuf=yr,mo,day,hr,mn,sec
- RTC=CPs since 1/1/90
- **ftimeout** *os/callout.c* set programmable clock to 1 sec future

IOS E only: **utcsetdate**: *c1/io/utc.c* utc driver date setup
PCI(HZ) Run PCI at 1 second in cpu–0
_CCI
CRAYC90) only: _ECI enable channel interrupts
setxp0error *c1/md/mfinit.c* Initialize error xp at 0 (to go to `zerotrap`)
clmask = 1 bits for "user" clusters 2–n
UTRACE 'GOU' *trace "go unix!"*
gounix *c1/md/setstk.s* call to finish startup
- Reset stack pointers B066 p0stack base, B067 top of stack, B002=0
- trace "GOU" *gounix*
- **umain** *c1/os/main.c* (continued next page)

park *c1/md/setstk.s*
- **SETGBT**
  - set B "global" registers for "this" CPU
  - *see diagram for CPU 0 at the start of mfstart*
- XA = pw.xpux
- CIP clear any IPI
- PCI clear PC – disable this CPU's PC
- save "INIT" in "its" PWS area (tell CPU 0 to go)
- SEMTSQ (wait on OSLOCK)

========== spin wait ================

- set stack pointers (B066/B067) to
  temporary stack area $STACKP in *c1/md/setstk.s*
- **mcpu** *c1/md/icpu.c* returns via longjmp later
  - set into system clustor (1)
  - connect it to its idle (created by umain/idle[0])
  - setperf *c1/md/perfmon.s* initialize HPM regs
  - SetMachineInfo *c1/sys/machinfo.h* build
    default machinfo data
  - if 'REQS' (first start) longjmp to save area (idle)
- Jump **master** (on 2nd return?)

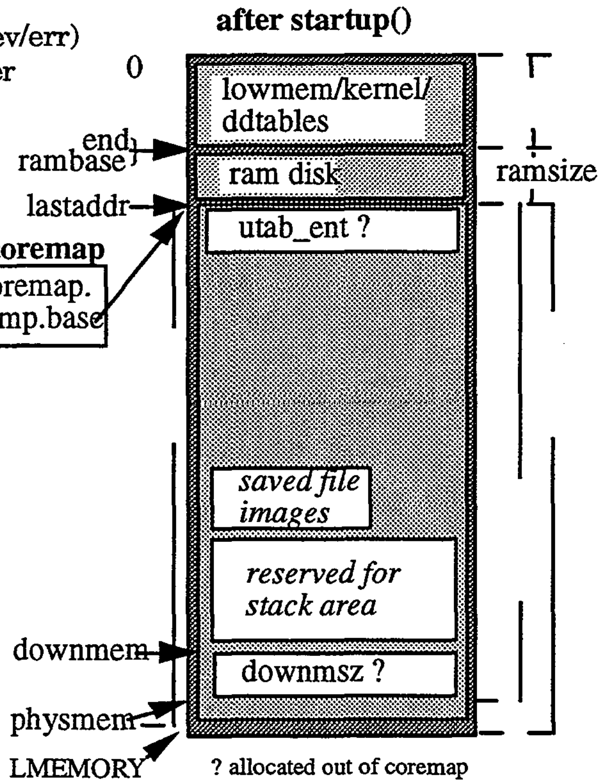
umain()logic

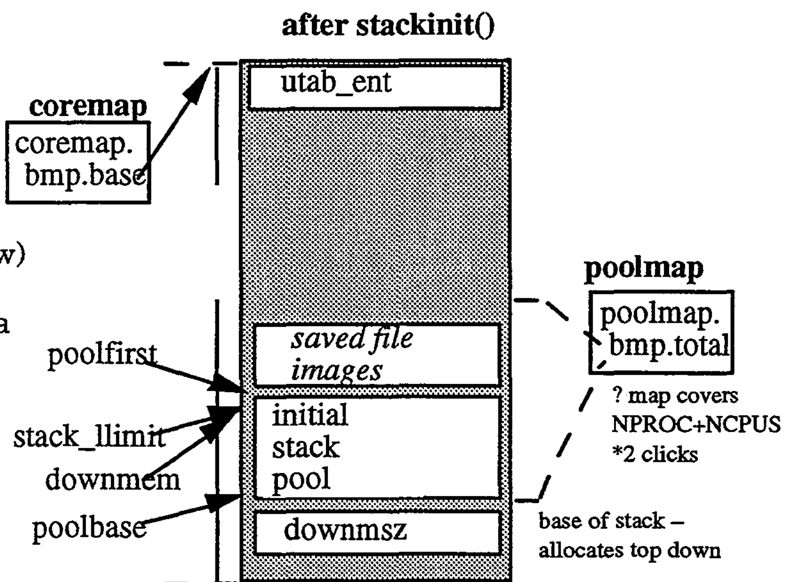
The diagrams on the right illustrate memory initialization by `umain`.

- Flags `init` "initialization in progress".
- Initializes error log and panic buffers.
- Calls `startup` to:
  - Adjust memory pointers for table usage.
  - Initialize user memory `coremap`, swapper, SDS, and BMR memory maps.
  - Allocates and initializes user table entry point `utab_ent` for dump processing.
- Calls `initsema` to initialize thread control fields for proc[0].
- Initializes proc[0] share pri., limit node, `proc`, and `pcomm` fields: proc[0] now logically connected to CPU 0.
- Allocates initial kernel stack pool from user memory `coremap` and adjusts user memory valuess downward. (See "Kernel Stack Management" for detail).

umain()logic

umain *c1/os/main.c*
init++
err_init *c1/io/errlog.c* initialize errlor log (/dev/err)
panicinit *c1/os/panic.c* initiialize panic buffer
startup *c1/md/machdep.c*
copy "ram disk" down next to kernel
maxmem=physmem–(end+ramsize)
initialize memory bitmap
coremap *area for user processes*
downmem=maxmem,
downmsz=hardmem–downmem
adjust coremap for downmem
initialize swapper bit map swp_map
initialize SDS map sdsmap
initialize BMR map bmrmap
allocate (coremap) utab_ent *for mfdump*
up=&proc[0], upc=&(proc[0].pcomm)
initialiaze proc[0] data fields
link proc[0] to pid hash queue
initsema *include_sys_semmacros.h*
initialize pc_sema for proc[0]
MaxSharePri=1.0
setup lnode[0] (system) lnode[1] (idles)
connect this CPU to proc[0]
ucinit *c1/os/main.c*
initialize proc[0] pcomm data areas
stackinit *c1/md/machd.c*
poollen=MAXCPUS+20 (MEMKLICKs)
allocate stack area poollen units
(at top of coremap – after downmem)
initialize poolmap:
NPROC+NCPU*2 units (max it can grow)
poolbase = high (click) address of stack area
poolfirst = low (click) address of stack area
p_stack (proc[0] stack) = p0stack
decrease maxmem by stack pool size
adjust downmem address down
by stack pool size
stack_llimit = low word
address of stack area
(return from stackinit -
umain continued next page)





umain()logic (continued)

- Initializes (possibly allocates) the following kernel memory areas and tables: (Details of these tables can be found with corresponding topics in this manual).
  - Security log buffers (if secure flag on).
  - Memory block table used by swapper.
  - Callout table used by CPU management.
  - Virtual File System function entry table.
  - Character (terminal) buffer areas.
  - System buffer management:
    - ◊ Buffer headers.
    - ◊ Cache buffer blocks.
    - ◊ Cache hash table headers.
    - ◊ Async (uoi) headers.
    - ◊ Exec map pool and bit map area.
    - ◊ Ldcache headers and bitmap areas.
  - Quota table.
  - File table.
  - Restart table and buffers.
  - Communication tables.
  - Proc table entries (excluding proc[0] already in use).
  - File locking tables.

umain()logic (continued)

umain *c1/os/main.c* (continued)
- **secinit** *c1/os/secure.c*
  - set secure_sys=1
  - **slginit** *c1/os/slogext.c*
    - allocate security log pseudo device buffer and flags
    - If security logging is enabled, issue initial system startup record
    - if secure system, set proc[0] security fields
- **memblk_init** *c1/os/sched.c*
  - link each proc table entry to a memblk entry
  - link each text table entry to a memblk entry
- **memblk_enq** *c1/os/sched.c*
  - initialize memblk entries
- **callinit** *os_callout.c*
  - initialize callout table entries
- **vfsinit** fs_vfs.c
  - initialize virtual file system (vfs) init function entries
- **cinit** *c1/io/clist.c*
  - link all character buffer blocks (cblocks) on cfreelist
- **binit** *c1/os/bio.c*
  - allocate (coremap) cache buffer headers (bufhd)
  - allocate (coremap) cache buffers (buffers)
  - allocate (coremap) cache hash buffer headers (hbuf)
  - initialize cache (MCACHE) allocation bit map
  - allocate (coremap) cache hash headers (hblks)
  - allocate (coremap) and clear uio table (uio_head)
  - *IOS B/C/D:* allocate (coremap) disk spare table (spare)
  - allocate (coremap) and clear exec hold arg. area (execbase)
  - initialize exec bit map
  - decrease maxmem and usrmem by above amounts
  - **schedv_adjust** *c1/os/sched.c* (no function here)
  - link cache and physical buf header to free lists
  - link uio and aio entries to free lists
  - *IOS B/C/D:* link and initialize spare table entries
  - **ldch_init** *c1/io/ldcache.c*
    - allocate (coremap) ldch headers (ldchlist)
    - allocate (coremap) and initialize ldch bit map (ldch_corebits)
- **qinit** *c1/os/quota.c*
  - initialize quota table and link on free list
- **finit** *c1/os/fio.c*
  - link file table entries to free list
- **restartinit** *c1/os/restart.c*
  - allocate (coremap) and initialize restart buffers (resinfo)
- **comminit** *c1/io/commsubr.c*
  - calculate size of communications tables
- **forkinit** *c1/os/fork.c*
  - link proc table entries (proc[1] through proc[NPROC]) to availproc
  - link proc[0] to allproc list
- **flckinit** *os_flock.c*
  - link flox table entries to free list

(continued next page)

umain()logic (continued)

- Partition memory for compatability mode.
- Message buffers for telnet, etc.
- System call stastics `sysent` table.
- Semaphore lock rule table (if configured).
- File system log buffers.
- Sidedoor buffers (if SDS configured).
- Tape daemon tables.
- Data migration tables.

Following the initialization of tables `umain` creates the system process as described on the next set of pages. The logic of `umain` continues after an overview of this activity.

umain()logic (continued)

- **allocinit** *c1/os/malloc.c*
  - partition memory for compatibility mode
- **minit** *tcp/kern/uipc_mbuf.c*
  - allocate (coremap) message buffer headers (_mhbase)
  - allocate (coremap) message buffers (_mdbase)
- **Netinit** *tcp/kern/net_subr.c*
  - **mbinit** *tcp/kern/uipc_mbuf.c* initialize mbufs
  - **configure** *tcp/kern/net_subr.c* Attach all the ethernet interfaces
  - **Dnet_attach** *tcp/kern/net_subr.c*
    - **ifinit** *tcp_net_if.c* initialize network interface table
    - **domaininit** *tcp_kern_uipc_domai.c* initialize tcp domains
    - **loattach** *tcp/net/if_loop.c* loopback interface driver
- **sysentinit** *c1_os_sysent.c*
  - initialize sysent table
- **init_lockrules** *c1/os/subr.c*
  - initialize kernel semaphore lock rule table
- **fslginit** *c1/io/fslog.c*
  - allocate(coremap) the file system log pseudo device buffer (fslgp)
- *if sdsunits*
  - **sideinit** *c1/io/sidedoor.c*
    - allocate (coremap) sidedoor buffers (sidebuf)
- **tpdinit** *c1/io/etpd.c*
  - initialize pointers to tape daemon tables
  - allocate (coremap) tape table storage area (stortab)
- **miginit** *c1/io/mig.c*
  - initialize queues for migration devices
  - *(umain continued later in this section)*

This page used for alignment

sysproc() routine

Summary

Function umain() (proc[0]) creates the "other" system processes using sysproc(). Function sysproc() is a form of fork/exec called by the kernel during startup to create the following system processes (NCPU is number of configured CPUs):

- init – proc[1]
- idle(0 thru NCPU); proc[2] thru proc[NCPU–1]
- esdpulse – proc NCPU]
- utility – proc[NCPU+1]

Creating system processes

The following pages illustrate how sysproc() creates these system processes. Note that entering this logic, only CPU 0 is executing, the other CPUs are spinning on OSLOCK in park. The two illustrations show:

- The logic of sysproc().
- proc[0] creating the system processes using sysproc()..

sysproc() example

sysproc() is called with the name of the new process, flag "FORK_FORK", size of newly created process, address of new process's program, and the index to the lnode to accumulate usage history.

The diagram on the right shows the basic logic of sysproc(). The diagram assumes the process proc[p] has called sysproc() to create a new process of name *name*, code address *codeaddr* and size *memsize*. The "parent" process image, with its stack, is shown on the lower right of the facing page. In the case of proc[0] making the call, the stack is in p0stack, otherwise it is in a unique area in the stack pool.

1. sysproc() calls newproc(), the "body" of the kernel's fork function.
2. newproc() gets the first available proc table slot for the new process. The newly starting system has only proc[0] used so far. Proc[1], proc[2] and so on, will be allocated in order. The child gets a new process ID and the parent's proc data is copied to the child' proc slot.
3. The parent saves its V registers and clusters in its process image so they may be inherited by the child (via the memory copy).
4. setjump() is called to save the current stack pointers in the parent's kernel register save area. This, in effect, saves the parent's stack (in the context of doing newproc() called by sysproc() called by umain()). setjump() returns zero (0) to its caller.
5. The return value from setjump() is tested by newproc(). The "false path" of logic finishes the parent's part of the fork.
   - Allocates a stack from the stack pool for the new process.
   - Allocates memory (coremap) for the new process (size of parent).
   - Copies the parent image to the new area.
   - Copies the parent stack to the child's area (its pointers are relocated when it is connected).
   - Places the child's proc table entry on the runq, it's a candidate to be connected by a CPU.
   - Returns zero (0) indicating "parent". The parent's side of the fork is basically completed.
6. The newproc() return is tested by sysproc(). The "parents's" logic (0) finishes its fork logic by marking the new process as loaded in memory, sets its priority.
7. The return of zero (0) indicates the parent returning from sysproc().
8. The new child process was left on the runq. Eventually a process (for example, proc[0]) will call swtch() to give its CPU to another process.
9. The new process is selected and "connected" to the CPU via a call longjmp().
10. longjmp()
    - Restores the new process's registers (copy of the parent's)
    - Restores the pointers to the stack (the copy of the parent).
    - Returns one (1) to the caller (actually the caller of setjump()).
11. The "true" test indicates to newproc() that it is the child executing newproc().
12. In newproc() the child adjusts the inherited XP values. newproc() returns one (1) as child. The "fork" is basically completed.
13. The child executes the "exec" part of the logic.
    - Assigns and fills out the caller specified lnode
    - Copies the caller specified name to the proc table entry.
    - Adjusts its memory (code/data) to caller specified memsize.
    - Copy the caller specified binary code codeaddr to the child's program space.
    - Adjusts the XP to the new (base/limit) values, etc.
14. The child's return of one (1) indicates to umain() this is the child's logic thread (will be tested by caller).

sysproc(*name,fork_flags,memsize,codeaddr,lnodeindex*)

1 **newproc** (*fork_flags, ...*)

2
get a proc slot for child
compute a new pid for child
zero child's proc table fields
copy "parent" proc to "child"
link to proc management chains
(eg. availproc–>allproc)

3 save the parent's V's and clusters

setjmp

4
save parent's stack pointers
return 0

any process calls **8**

swtch
selects process from runq **9**
longjmp (saved stack) context
restores new process's regs
restores new process's stack
10 return 1

return 0?

return 1? **11**

5
allocate (stackpool) stack for child
allocate memory (coremap) for child
copy parent (image) to child's
copy parent stack to child's
place child on the runq
return 0

sureg
build process's XP
return 1
12

return 1? **13**

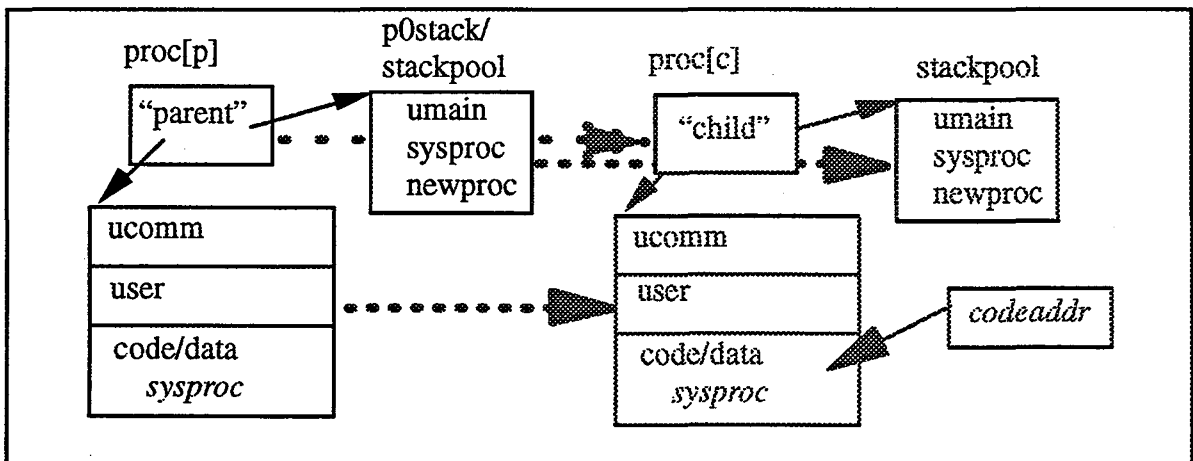
return 0?

6
mark child loaded in memory
set child's sharepri=MAXSHAREPRI
return 0 (as original thread / parent)

7

assign and fill out lnode*[lnodeindex]*
copy *name* to child proc
adjust process to *memsize*
copy *codeaddr* (*memsize*) to child's memory
adjust XP to new memory address
return 1 (as new process / child)

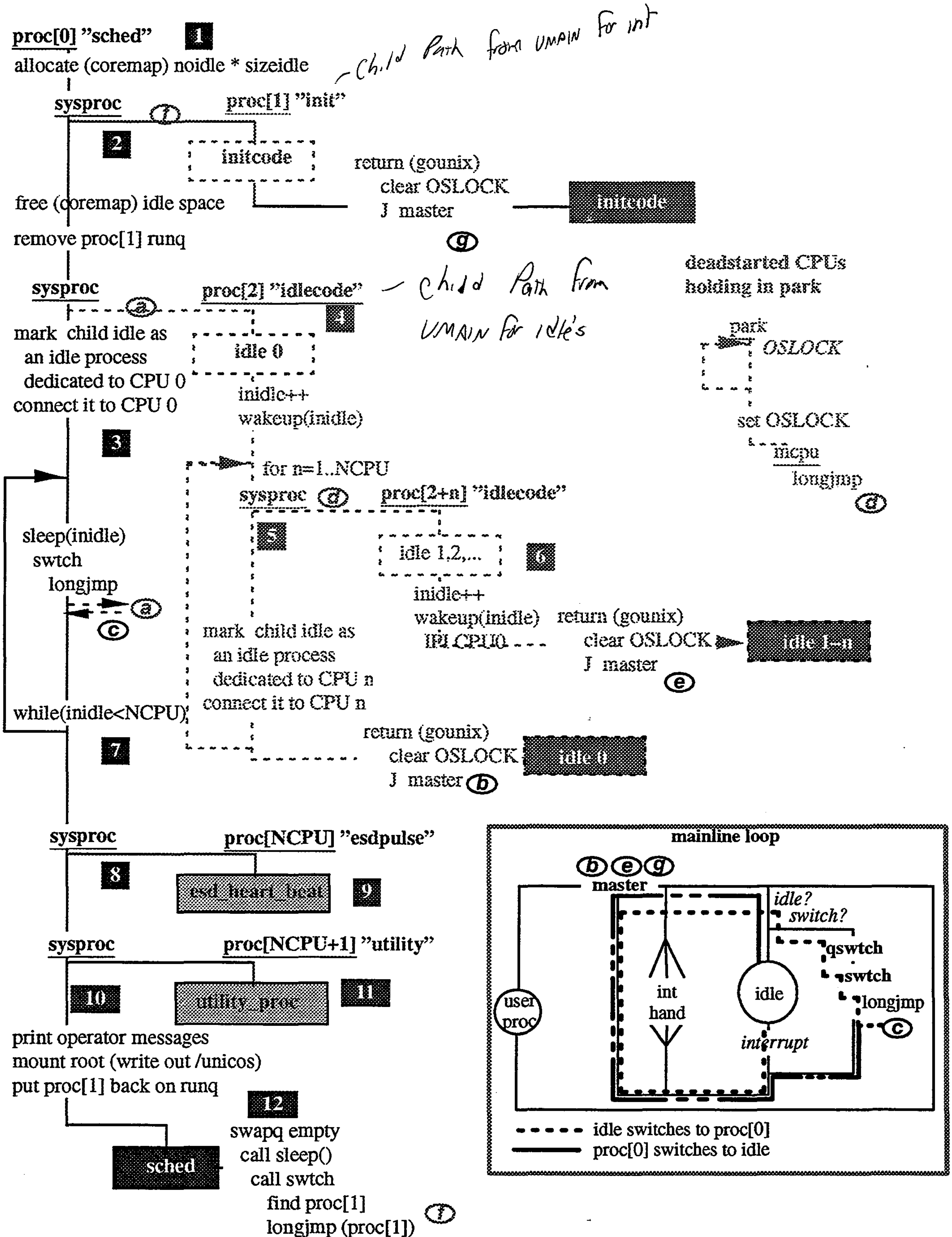
14



`sysproc()` routine

Creating system processes

1. Proc[0] allocates enough memory to hold all the idles (just after `utab_ent`). `sysproc()` is called to create the init process proc[1]. Recall that at the moment this process is simply a copy of proc[0] queued on the `runq`.
2. `umain()` frees the memory held for idles and removes proc[1] from the `runq` (it can't run until after the idles are created and root is mounted).
3. `umain()`
   - Creates the first idle process (proc[2]), leaving it on the `runq`.
   - Flags the process as "IDLE", dedicates it to CPU 0, and connects it to CPU 0 (PWS table).
   - Enters a while loop, calling `sleep()` until all idles have started. `Sleep()` calls `swtch()` which selects another process to run (only candidate proc[2] idle 0) and
   - Executes longjmp to `umain()`, and continues in child logic path *(a)*.
4. Idle 0 counts that another idle has started and awakens proc[0] (it will run when CPU 0 actually exchanges to the idle process).
5. Idle 0 creates the other idles.
   - Enters a `for` loop
     - Creates each other idle process with `sysproc()` and marks it as "IDLE"
     - Dedicates it to the corresponding CPU,
     - Connects it to the CPU (PWS table).
   - After the last process is "forked", idle 0 (proc [2])
     - Returns (in the context of `umain()`)
     - Returns to `gounix()`
     - Clears OSLOCK (this allows one of the parked CPUs to leave `park()`).
     - Jumps to `master` in the mainline loop *(b)*. In the mainline the CPU would "try to" exchange to the idle code but proc[0] is on the runq at a better priority. `swtch()`
       - ◇ Disconnects idle 0 proc[2]
       - ◇ Selects proc[0] to execute.
       - ◇ Long jumps to proc[0] (in sleep) *(c)*.
   - The awakened proc[0] calls `sleep` again in the while loop since not all idles are started. CPU 0 will disconnect from proc[0] and reconnect to proc[2] as idle.
6. Clearing OSLOCK frees one CPU (at a time) from `park()`. The CPU
   - Sets OSLOCK for itself
   - Long jumps to its corresponding idle (finish fork/exec in `sysproc()`) *(d)*.
   - Counts another idle started.
   - Awakens proc[0], sends IPI to CPU 0. (swtch to umain *(d)* & back until inidle>NCPU).
   - The new idle return to caller `gounix()` clears `OSLOCK`, jumps to `master` *(e)*, and exchanges to the idle loop.
7. When the last idle awakens proc[0] drops out of loop. `umain()` continues.
8. `umain()` creates process `esdpulse` used by the shared file system.
9. `esd_heart_beat()` is a function in the kernel executing as an infinite loop. It awakens on a timed interval, polls the esd device, performs an sfs service, and calls sleep.
10. `umain()` creates process `utility`.
11. `utility_proc()` is an infinite loop like the `esdpulse()` that performs a service function in the kernel function and calls `sleep()`.
12. `umain()` prints the "welcome to UNICOS" operator messages, mounts the root file system, and puts proc[1] back on the `runq`. The last step of `umain` is to call the swapper `sched()`, another infinite loop function in the kernel. `Sched()` will find no processes to swap in so it calls `sleep()` which calls `swtch()`, which now finds proc[1] on the `runq`. The last step shown is the CPU calling longjmp to connect `init` proc[1] *(f/g)*. More detail about this last item (12) is on the following pages.


proc[0] "sched" 1
allocate (coremap) noidle * sizeidle
- Child Path from UMAIN for init
sysproc
f
proc[1] "init"
2
initcode
return (gounix)
clear OSLOCK
J master
initcode
free (coremap) idle space
remove proc[1] runq
g
deadstarted CPUs
holding in park
sysproc
a
proc[2] "idlecode"
- Child Path from UMAIN for idle's
4
mark child idle as
an idle process
dedicated to CPU 0
connect it to CPU 0
idle 0
park
OSLOCK
inidle++
wakeup(inidle)
set OSLOCK
incpu
longjmp
d
3
for n=1..NCPU
sysproc
d
proc[2+n] "idlecode"
5
idle 1,2,...
6
sleep(inidle)
swtch
longjmp
a
c
inidle++
wakeup(inidle)
IPI CPU0
return (gounix)
clear OSLOCK
J master
idle 1–n
e
mark child idle as
an idle process
dedicated to CPU n
connect it to CPU n
while(inidle<NCPU)
7
return (gounix)
clear OSLOCK
J master
idle 0
b
sysproc
proc[NCPU] "esdpulse"
8
esd_heart_beat
9
sysproc
proc[NCPU+1] "utility"
10
utility_proc
11
print operator messages
mount root (write out /unicos)
put proc[1] back on runq
12
swapq empty
call sleep()
call swtch
find proc[1]
longjmp (proc[1])
f
sched
mainline loop
b e g
master
idle?
switch?
qswtch
swtch
longjmp
c
user proc
int hand
idle
interrupt
idle switches to proc[0]
proc[0] switches to idle

umain (continued)

After the system processes have been created (forked but not necessarily entered yet) umain() continues startup action by doing the following:

- Sets proc[0] (future sched()) size to the area between its ucomm and the first idle.
- Performs "down CPU" action on any CPU configured down at startup. These CPUs will be interrupted from the "normal" idle and enter a down CPU idle (see topic in "Mainline Loop" chapter.
- Initializes directory name lookup cache with function dnlc_init().
- Initializes (opens device) and mounts the root file system
- rootdev specified in the startup parameter file
- Function vfs_mountroot() calls nc1mountroot().
- Writes the "file images" to disk (on root)
- (See topic in "File System Management" chapter)
- Functions ipi3_init() and hpi3_init() initialize IPI and Hippi device drivers.
- On a secure system the administrator console (/dev/console) security level and compartments are set.

(Continued on next page)

umain (continued)

proc[0] (sched) size = ucomm area to first idle

print "welcome on" messages

down any CPUs configured down in the param file
(flag it's PWs entry PW_DOWN and PW_STOP,
send IPI to downed CPU,
it will leave "normal" idle and enter a diagnostic idle loop)

dnlc_init *fs/dnlc.c*
initialize directory name lookup cache table `ncache` *dnlc.c*

vfs_mountroot *fs/vfs.c mount the root file system (detail "File System Management")*
nc1mountroot *fs/nc1/nc1vfsops.c*
open root device:
I/O routines call `sleep()`
CPU disconnects from proc[0] and connects to idle
the I/O interrupt awakens proc[0] and interrupts the
CPU out of idle, technically any CPU may pick up
from here (not just CPU 0)
the open initializes flaw information for the device
open the swap device
if swap weight not a multiple of the swap device NOTE:
allocate RAM ldcache for the swap device
(well formed to largest disk I/O unit in device slice group)
if sds_conf.length != 0
open sds device
mount the root file system
IOS B/C/D: check for presence of backdoor channel
wakeup(rootdir) – (shouldn't do anything)
IOS B/C/D:
write `/IOS-param` from memory image of param file
write `/CONFIGURATION` from memory image of config file
write `/etc/setdev` from memory image of mknod script
write out `/unicos` file (if RAM root write only symbol table)
get vnode for "/"
set proc[0] uc_rdir = "/" vnode (root directory)
set proc[0] uc_cdir = "/"vnode (current directory)

ipi3_init *c1/io/ipi3.*
initialize ipi3 driver counters, limit, and traces

hpi3_init *c1/io/hpi3.c*
initialize hpi3 driver counters, limit, and traces

if secure system
secure_init *c1/os/secure.c*
set administrator console security level and compartments

locate /dev/null device number for restart logic

umain() (continued)

Function umain() finishes startup action by doing the following:

- Sets the name of proc[0] to "sched" (the swapper)
- Partitions the swap device (by slice makeup) in swap_init().
- Sets initial swapping tuning (schedv) values with schedv_init().
- Sets ininit to zero indicating initialization (nearly) complete.
- Initializes the "target machine" table in function targinit().
- Sets kernel flow trace flag "on" if FLOWTRACE configured.
- Enters a time event into the callout table indicating that lsp_monitor() should be executed every 2 seconds. This function checks LOWSP channel time–outs and requests IOS packet retransmissions if detected.
- Puts proc[1] back on the runq. It can run now.
- If any CPUs were marked down send operator message with this information.
- Calls function sched()

The swapper logic is entered at this time. sched() is basically an infinite loop (in the kernel) which checks on the memory situation, swapping processes out of memory and back in under a priority scheme. The swapper spends (hopefully) most of its time "sleeping", waiting for work to do. Each time it is awakened it performs its loop (swap cycle, and return to sleep). The first call of sched() results in the following:

  - Swapper finds the swap queue empty and calls sleep().
  - sleep() calls swtch() which disconnects this CPU from proc[0].
  - swtch() selects the best process to run (will be proc[1] now), restores its context, and long jmps to it.
  - The saved context of proc[1] sends the CPU to the "child" side logic in newproc() called by sysproc() called by umain().

- The "child" (return value from setjmp = 1) completes the creation of proc[1] and returns (all the way back to gounix).
- Assembler routine gounix resets the stack to the base umain frame, sets the single thread lock OSLOCK, and jumps to master. The CPU is in the kernel mainline loop.
- Logic in the mainline exchanges to the user process proc[1].
  - The code for initcode() consists only of an exec(2) system call, protocol:
    - ◇ Address of the calling parameters (path name) in S1 (/etc/init)
    - ◇ Number of function to perform (EXEC)
    - ◇ Exchange to the kernel
  - The initcode program exchanges immediately back to the kernel which performs the exec(2).
    - ◇ Adjusts the memory area to accommodate the "real" init binary.
    - ◇ Reads the init(8) (/etc/init) binary into the code/data area.
    - ◇ Exits the kernel – enters the user program at the a.out entry point.

UNICOS is now running, in init(8) in single user mode.

The kernel is fully functioning now, all other processing is "user" processing by init(8) and it's child processes.

umain (continued)

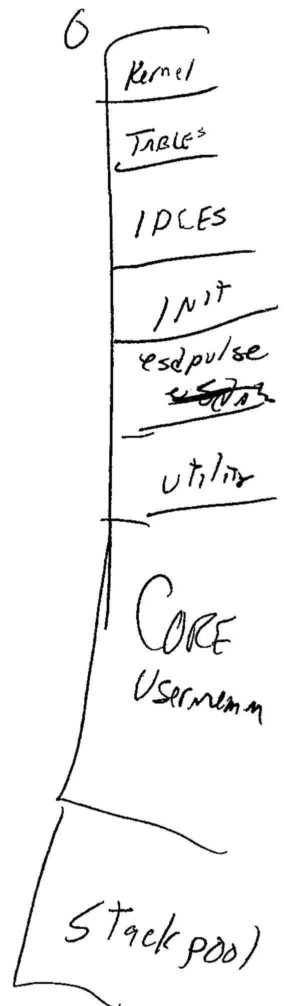
set proc[0] name to "sched"
sysmem = memory from 0 through last system proc (`utility`)
usrmem = what's left (sysmem to stack pool, file images can be reused now)
swap_init *c1/io/swap.c*
partition the swap device
schedv_init *c1/os/sched.c*
set initial swapper tuning values
ininit=0 *mark init done (well almost)*
targinit *os/target.c*
set host machine values in target table
set kernel flow trace flag (if FLOWTRACE defined)
lsp_monitor *c1/io/epack.c*
timeout *os/callout.c*
enter **lsp_monitor**() on timer queue (2 sec.) to monitor LOWSP chan timeout
put proc[1] (init) back on runq
if any CPUs marked down
print operator warning message

sched() *c1/os/sched.c*
swapq empty ?
sleep()
swtch
CPU connects to proc[1] (init) – **longjmps to ...**

resume at newproc() - child logic of proc[1] init
returns to sysproc() then ***umain***
set current directory to root directory
set OSLOCK (single thread until enter mainline)
return(1) to `gounix`
gounix (continued)
clear OSLOCK
reset stack to first (umain) frame
jump to master

master exits kernel / enters user process (**initcode**)
s1="/etc/init" *initcode set up exec system call*
s0=$EXEC
EX
init exchanges back to kernel to do `exec` system call
exec() *c1/os/exec.c*
expand code/data area of proc[1] to init's size
load (read) /etc/init binary into memory area
return to user (at init's entry point)

UNICOS is started – system executing `init(8)` in single user mode



Central memory sizes

LMEMORY: value of MEMORY truncated to a ba/la boundary: `sys/machd.h`

physmem: Physical memory available; for example, addressable by the kernel.
Compiled as `LMEMORY`.
Reset with `MEMORY =` *words* startup parameter file directive.
Adjusted down to iba/dba boudary.
Changed by `chmem`(2).

ddtbase: Base of device table area (also `pdummy`).

end: End of the kernel's non-malloc()'d space (a click boundary).

rambase / ramsize: If used, the ram disk would start and end on click boundaries.
Space is allocated only if the parameter file configures disk(s) of type DDRAM.

utab_ent: Pointer to the first table the kernel `malloc()`'d during startup.

hardmem: Memory which is or could be made available to `malloc`() (in `coremap`).
Adjusted by `startup`() to the `physmem - (end + ramsize)`.
Used to initialize `coremap` (so that it can grow later).

poolbase: Pointer (logical) base of kernel stack pool.
Adjusted to nearest `MEMKLICK`.

stack_llimit: Current (logical) top of the stack pool.
Allocated out of `coremap`.
† Moves up and down as stack pool grows (down) or shrinks.

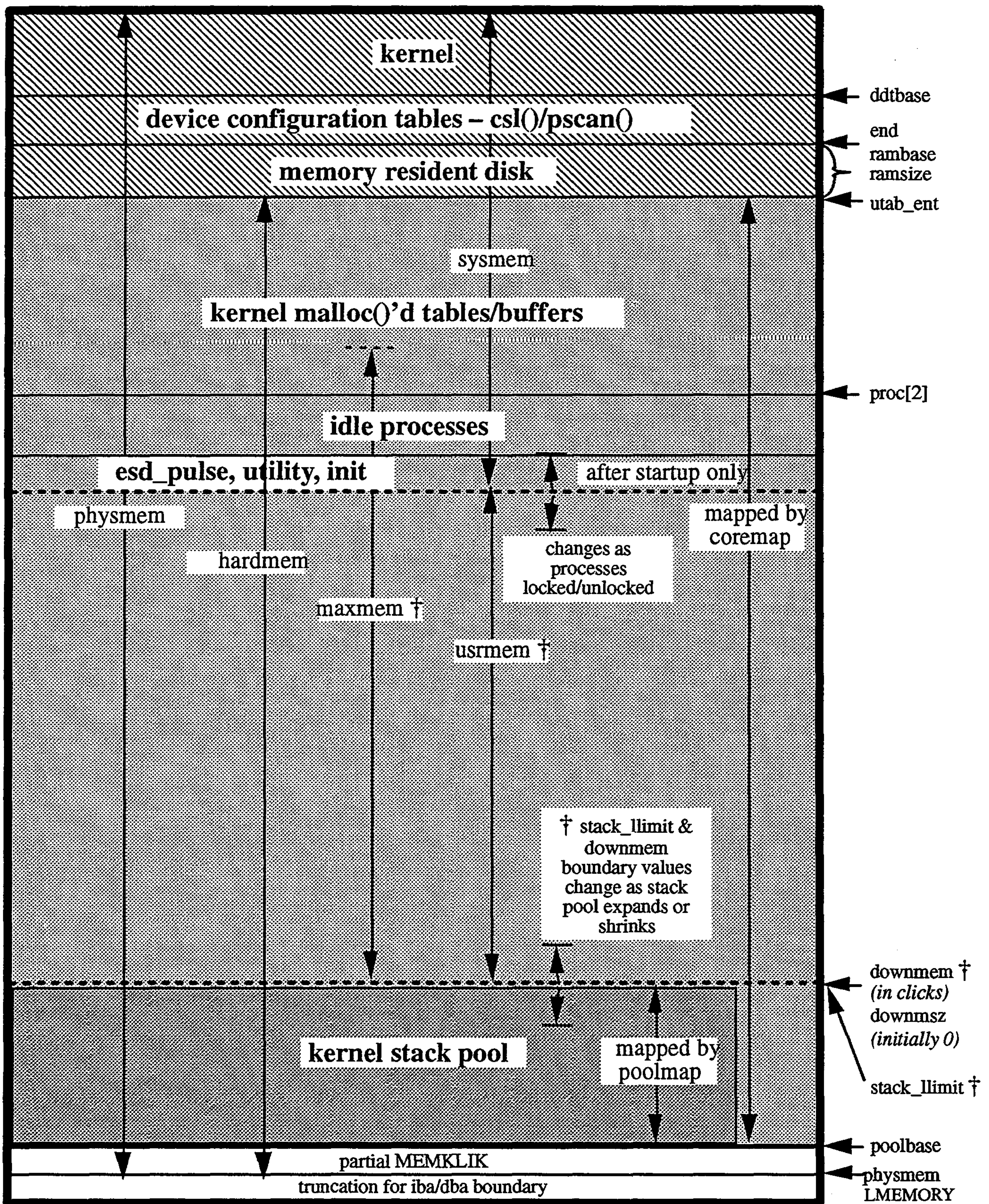
downmem / downmsz: Downed memory address / size (in clicks).
Limit of usable memory for user processes.
Same as `stack_llimit` but in clicks.

maxmem: Theoretical area for processes.
Initially `physmem - end - ramsize - poolsize - malloc`()'d tables.
Note, several `malloc`'d kernel tables are not deducted from it so it is a bit too big.
Used mainly in calculating accounting record memory integrals.

sysmem: System memory (kernel plus system tables plus system processes).
At deadstart, size of the binary.
Set to the address of remaining `coremap` memory just before entering `init`.
Increases each time a process `plock`(2)'s in memory and decreases when the lock is freed.
Readable via `sysconf`(2).

usrmem: Currently available user memory.
The area between the dynamically changing `sysmem` and `downmem`.

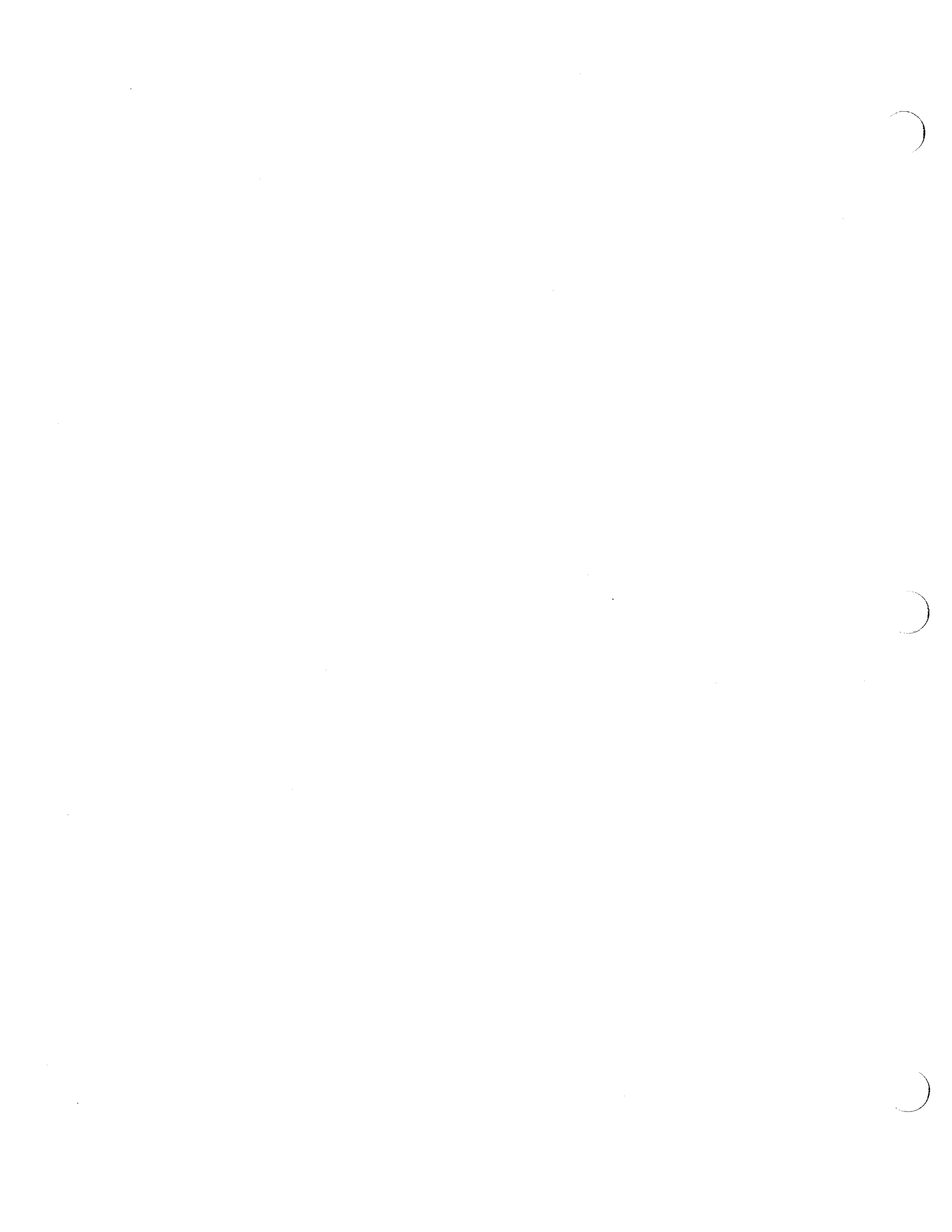
Central Memory Sizes



This page used for alignment

Contents

Kernel Mainline [4] 4–1
Objectives 4–1
Kernel mainline overview and mainline detail diagram 4–2
Mainline outer loop 4–8
Kernel entry 4–8
Initial saves and updates 4–8
immtrap - Trapping monitor mode interrupts 4–14
rpescrub and immrpe 4–16
Interrupt handler selection 4–18
Idle processes 4–22
Idle process selection 4–22
Idle process - idle CPU 4–23
Idle process - down CPU 4–24
giveup() and idler 4–26
Process selection 4–28
Signals 4–30
Signal detection 4–30
issig() - Kernel's test for a processable signal 4–32
Catching a signal 4–34
Signal data structures 4–37
Library routines words 4–40
Kernel signal processing overview 4–42
Library signal processing overview 4–45
Kernel exit 4–47
Mainline inner loop - Interrupt handlers 4–51
usrnex - User normal exit (system call) 4–51
System entry table 4–53
usrioi - I/O interrupt 4–60
LOWSP channels 4–62
User error interrupts 4–64
usrfpi - User floating-point interrupt 4–64
usrore - User operand range error 4–64
usrpre - User program range interrupt 4–64
usrbpi - User breakpoint interrupt (C90 only) 4–64
usreex - User error exit 4–64
usrdli - User deadlock interrupt 4–66
usrdli logic 4–68
usrpci - User programmable clock interrupt 4–70
usrmei - User memory error interrupt 4–72
usrmcu - User maintenance control unit 4–76
usrrtm - User real-time interrupt 4–76
usripi - User interprocessor interrupt 4–78
usrrpe - User register parity errors 4–80
usrmii - User monitor mode instruction interrupt 4–82



Kernel Mainline [4]

Objectives

After completing this section you should be able to:

- **Describe the general processing flow of the kernel mainline loop**
- **Define in detail the kernel's mainline logic loop**
- **Define in detail the kernel's interrupt handlers**

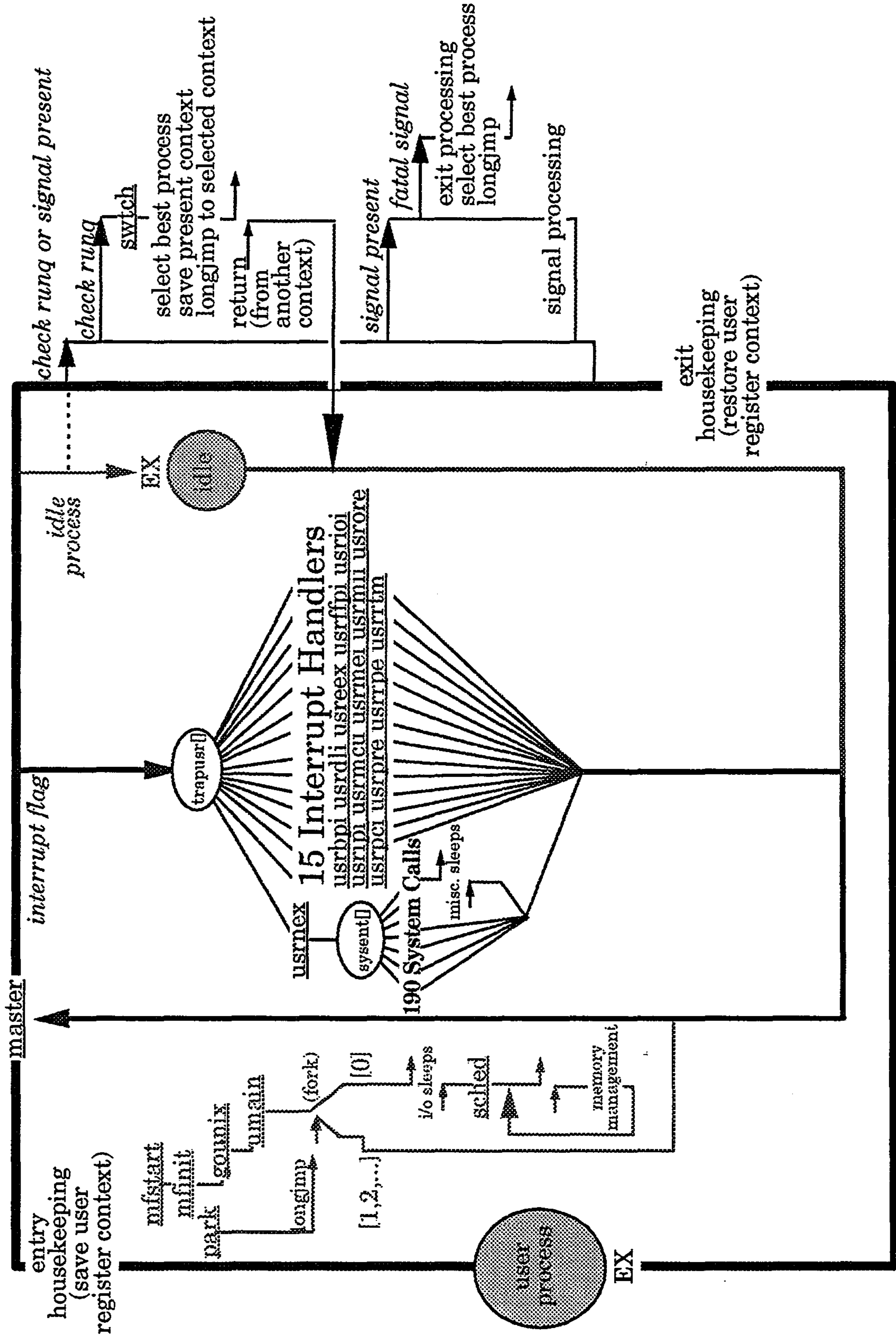
Kernel mainline overview and mainline detail diagram

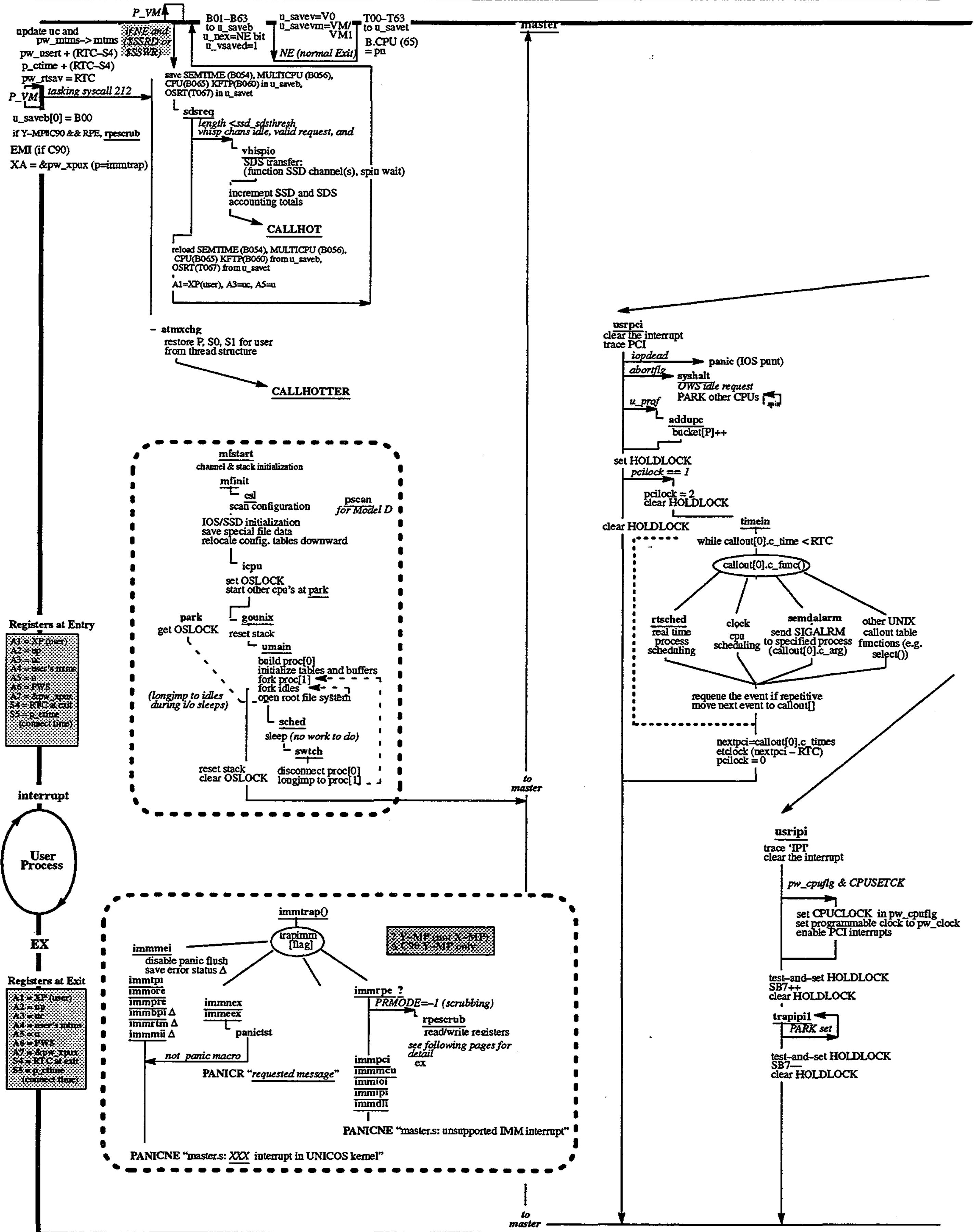
The diagram on the right shows an overview of the logic flow of the kernel's mainline. The term "Mainline" comes from the name of function `umain` in source file `c1/os/main.c`.

The two page diagram on the following pages is the same logic with more detail shown.

The remaining portion of the chapter describes in detail the mainline loop, dividing the logic shown in these diagrams into "modules", following the outer loop first, then discussing each interrupt hander in the inner loop.

Kernel Main Loop Overview




update uc and pw_mtms–> mtms
pw_usert + (RTC–S4)
p_ctime + (RTC–S4)
pw_rtsav = RTC
tasking syscall 212
u_saveb[0] = B00
if Y-MP/C90 && RPE, rpescrub
EMI (if C90)
XA = &pw_xpux (p=immtrap)
B01–B63 to u_saveb
u_nex=NE bit
u_vsaved=1
u_savev=V0
u_savevm=VM/VM1
NE (normal Exit)
T00–T63 to u_savet
B.CPU (65) = pn
master
sdsreq
vhispio
SDS transfer: (function SSD channel(s), spin wait)
increment SSD and SDS accounting totals
CALLHOT
A1=XP(user), A3=uc, A5=u
atmxchg
restore P, S0, S1 for user from thread structure
CALLHOTTER
mfstart
channel & stack initialization
mfinit
csl
scan configuration
pscan for Model D
IOS/SSD initialization
save special file data
relocate config. tables downward
icpu
set OSLOCK
start other cpu's at park
park
get OSLOCK
gounix
reset stack
umain
build proc[0]
initialize tables and buffers
fork proc[1]
fork idles
open root file system
(longjmp to idles during I/O sleeps)
sched
sleep (no work to do)
swtch
reset stack
clear OSLOCK
disconnect proc[0]
longjmp to proc[1]
to master
Registers at Entry
interrupt
User Process
EX
Registers at Exit
immtrap()
trapimm [flag]
immmei
disable panic flush
save error status Δ
immtpi
immore
immpre
immbpi Δ
immrtm Δ
immmii Δ
immnex
immeex
panictst
not panic macro
PANICR "requested message"
immrpe ?
PRMODE=–1 (scrubbing)
rpescrub
read/write registers
see following pages for detail
ex
immpci
immmcu
immioi
immipi
immdli
PANICNE "master.s: unsupported IMM interrupt"
PANICNE "master.s: XXX interrupt in UNICOS kernel"
usrpci
clear the interrupt
trace PCI
iopdead
panic (IOS punt)
abortflg
syshalt
OWS idle request
PARK other CPUs
u_prof
addupc
bucket[P]++
set HOLDLOCK
pcilock == 1
pcilock = 2
clear HOLDLOCK
clear HOLDLOCK
timein
while callout[0].c_time < RTC
callout[0].c_func()
rtsched
real time process scheduling
clock
cpu scheduling
semdalarm
send SIGALRM to specified process (callout[0].c_arg)
other UNIX callout table functions (e.g. select())
requeue the event if repetitive
move next event to callout[]
nextpci=callout[0].c_times
etclock (nextpci – RTC)
pcilock = 0
usripi
trace 'IPI'
clear the interrupt
pw_cpuflg & CPUSETCK
set CPUCLOCK in pw_cpuflg
set programmable clock to pw_clock
enable PCI interrupts
test-and-set HOLDLOCK
SB7++
clear HOLDLOCK
trapipi1
PARK set
test-and-set HOLDLOCK
SB7—
clear HOLDLOCK
to master

P_VM

update uc and pw_mims–> mims
if NE and $SSRD or $SSWR
pw_usert + (RTC–S4)
p_ctime + (RTC–S4)
pw_rtsav = RTC

B01–B63 to u_saveb
u_nex=NE bit
u_vsaved=1

u_savev=V0
u_savevm=VM/VM1
NE (normal Exit)

T00–T63 to u_savet
B.CPU (65) = pn

master

P_VM
tasking syscall 212
u_saveb[0] = B00
if Y-MP/C90 && RPE, rpescrub
EMI (if C90)
XA = &pw_xpux (p=immtrap)

save SEMTIME (B054), MULTICPU (B056), CPU(B065) KFTP(B060) in u_saveb, OSRT(T067) in u_savet

sdsreq
length < ssd_sdsthresh
whisp chans idle, valid request, and

vhispio
SDS transfer:
(function SSD channel(s), spin wait)

increment SSD and SDS accounting totals

CALLHOT

reload SEMTIME (B054), MULTICPU (B056), CPU(B065) KFTP(B060) from u_saveb, OSRT(T067) from u_savet
A1=XP(user), A3=uc, A5=u

atmxchg
restore P, S0, S1 for user from thread structure

CALLHOTTER

mfstart
channel & stack initialization
mfinit
csl
scan configuration
pscan for Model D
IOS/SSD initialization
save special file data
relocate config. tables downward
icpu
set OSLOCK
start other cpu's at park
park
get OSLOCK
gounix
reset stack
umain
build proc[0]
initialize tables and buffers
fork proc[1]
fork idles
open root file system
(longjmp to idles during i/o sleeps)
sched
sleep (no work to do)
swtch
reset stack
clear OSLOCK
disconnect proc[0]
longjmp to proc[1]

to master

Registers at Entry
A1 = XP (user)
A2 = up
A3 = uc
A4 = user's mims
A5 = u
A6 = PWS
A7 = &pw_xpux
S4 = RTC at exit
S5 = p_ctime (connect time)

interrupt

User Process

EX

Registers at Exit
A1 = XP (user)
A2 = up
A3 = uc
A4 = user's mims
A5 = u
A6 = PWS
A7 = &pw_xpux
S4 = RTC at exit
S5 = p_ctime (connect time)

immtrap()
trapimm [flag]
Y-MP (not X-MP) & C90 Y-MP only
immmei
disable panic flush
save error status Δ
immtpi
immore
immpre
immbpi Δ
immrlm Δ
immmli Δ
immnex
immeex
panictst
not panic macro
PANICR "requested message"
immrpe ?
PRMODE=–1 (scrubbing)
rpescrub
read/write registers
see following pages for detail
ex
immpci
immmcu
immioi
immipi
immdli
PANICNE "master.s: unsupported IMM interrupt"
PANICNE "master.s: XXX interrupt in UNICOS kernel"

to master

usrpci
clear the interrupt
trace PCI
iopdead
panic (IOS punt)
abortflg
syshalt
OWS idle request
PARK other CPUs
u_prof
addupc
bucket[P]++
set HOLDLOCK
pcilock == 1
pcilock = 2
clear HOLDLOCK
clear HOLDLOCK
timein
while callout[0].c_time < RTC
callout[0].c_func()
rtsched
real time process scheduling
clock
cpu scheduling
semdalarm
send SIGALRM to specified process (callout[0].c_arg)
other UNIX callout table functions (e.g. select())
requeue the event if repetitive
move next event to callout[]
nextpci=callout[0].c_times
etclock (nextpci – RTC)
pcilock = 0

usripi
trace 'IPI'
clear the interrupt
pw_cpuflg & CPUSETCK
set CPUCLOCK in pw_cpuflg
set programmable clock to pw_clock
enable PCI interrupts
test-and-set HOLDLOCK
SB7++
clear HOLDLOCK
trapipi1
PARK set
test-and-set HOLDLOCK
SB7—
clear HOLDLOCK

to master

Clear HOLDLOCK
Clear leftmost XP flag
A1 = &pw_xpux
S1 = XP flags

!ST6 && (CI > 0 (and not in ST1) or ST0 > 0)
Set IOI in S1 (else clear IOI in XP)

set HOLDLOCK

this cpu's bit set in ST7
Set ICP in S1
Clear ST7 bit

!pcilock && RTC > nextpci
Set PCI in S1
pcilock = 1

forcevector

B.UP (62) = pw_userp
B.UPC (61) = p_pc
B.U (64) = pw_useru
B.UC (63) = pw_ucomm
B.KFTP=0
T.TRACEM=tracemask
any xp flags set (bits in S1)

trapusr[] entry:
handler address
trace ascii | signal

B.SUBR (04) = trapusr[flag]
S1 = trapusr[] trace info / signo

/usr/src/lib/asdef/asdef.s
B.%STKCBP (02) = &u_stack "Current Base"
B.%STKCTP (66) = frame[1] "Current Top"
B.%STKATP (67) = limit "Absolute Top"

B.SUBR

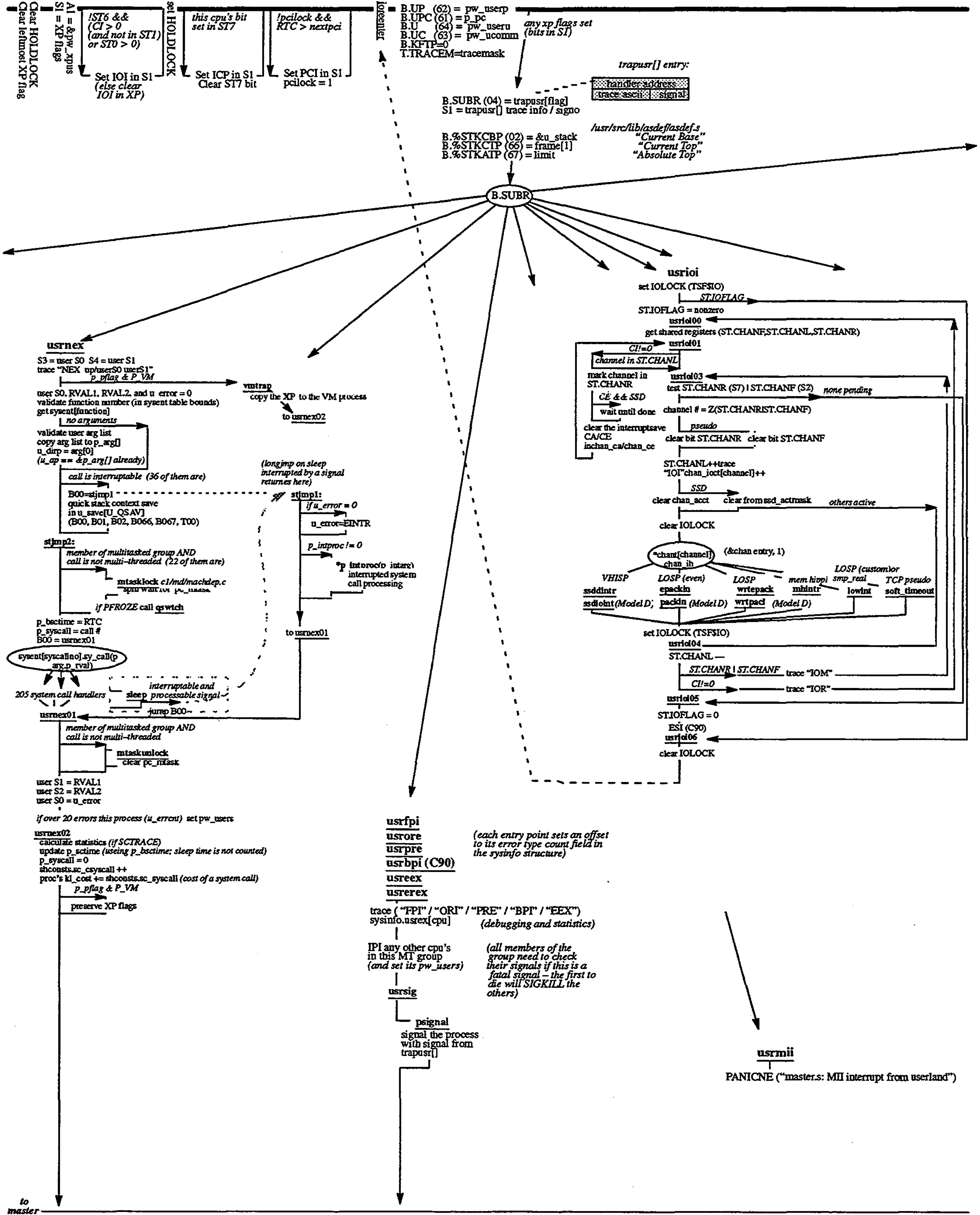
usrnex
S3 = user S0 S4 = user S1
trace "NEX up/userS0 userS1"
p_pflag & P_VM
vmtrap
copy the XP to the VM process
to usrnex02
user S0, RVAL1, RVAL2, and u_error = 0
validate function number (in sysent table bounds)
get sysent[function]
no arguments
validate user arg list
copy arg list to p_arg[]
u_dirp = arg[0]
(u_ap == &p_arg[] already)
call is interruptable (36 of them are)
(longjmp on sleep interrupted by a signal returns here)
B00=stjmp1
quick stack context save in u_save[U_QSAV] (B00, B01, B02, B066, B067, T00)
stjmp1:
if u_error = 0
u_error=EINTR
p_intproc != 0
interrupted system call processing
to usrnex01
stjmp2:
member of multitasked group AND call is not multi-threaded (22 of them are)
mtasklock c1/md/machdep.c
if PFROZE call qswtch
p_bsctime = RTC
p_syscall = call #
B00 = usrnex01
sysent[syscallno].sy_call(p arg,p rval)
205 system call handlers
sleep
interruptable and processable signal
jump B00–
usrnex01
member of multitasked group AND call is not multi-threaded
mtaskunlock
clear pc_mtask
user S1 = RVAL1
user S2 = RVAL2
user S0 = u_error
if over 20 errors this process (u_errcnt) set pw_users
usrnex02
calculate statistics (if SCTRACE)
update p_sctime (using p_bsctime; sleep time is not counted)
p_syscall = 0
shconsts.sc_csyscall ++
proc's kl_cost += shconsts.sc_syscall (cost of a system call)
p_pflag & P_VM
preserve XP flags

to master

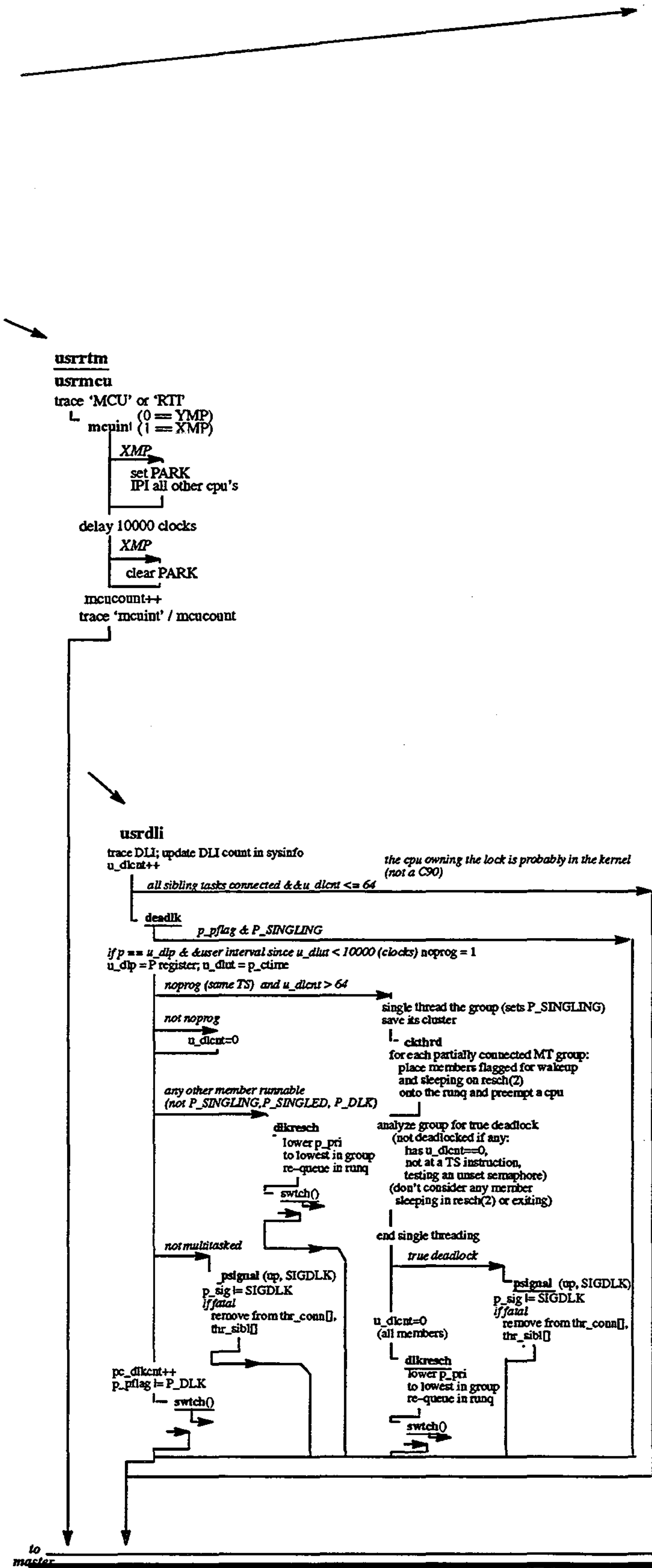
usrioi
set IOLOCK (TSFSIO)
ST.IOFLAG
ST.IOFLAG = nonzero
usrioi00
get shared registers (ST.CHANF,ST.CHANL,ST.CHANR)
usrioi01
CI!=0
channel in ST.CHANL
mark channel in ST.CHANR
CE && SSD
wait until done
clear the interrupt save CA/CE
inchan_ca/chan_ce
usrioi03
test ST.CHANR (S7) | ST.CHANF (S2)
none pending
channel # = Z(ST.CHANR|ST.CHANF)
pseudo
clear bit ST.CHANR clear bit ST.CHANF
ST.CHANL++trace
"IOI"chan_ioct[channel]++
SSD
clear chan_acct
clear from ssd_actmask
others active
clear IOLOCK
*chant[channel] chan_ih
(&chan entry, 1)
VHISP
ssddintr
ssdloint (Model D)
LOSP (even)
epackin
packin (Model D)
LOSP
wrtepack
wrtpaci (Model D)
mem hippi
mhintr
LOSP (customer)
smp_real
lowint
TCP pseudo
soft_timeout
set IOLOCK (TSFSIO)
usrioi04
ST.CHANL —
ST.CHANR | ST.CHANF
trace "IOM"
CI!=0
trace "IOR"
usrioi05
ST.IOFLAG = 0
ESI (C90)
usrioi06
clear IOLOCK

usrfpi
usrore
usrpre
usrbpi (C90)
usreex
userex
(each entry point sets an offset to its error type count field in the sysinfo structure)
trace ("FPI" / "ORI" / "PRE" / "BPI" / "EEX")
sysinfo.usrex[cpu]
(debugging and statistics)
IPI any other cpu's in this MT group (and set its pw_users)
(all members of the group need to check their signals if this is a fatal signal – the first to die will SIGKILL the others)
usrsig
psignal
signal the process with signal from trapusr[]

usrmii
PANICNE ("master.s: MII interrupt from userland")



B.UP (62) = pw_userp
B.UPC (61) = p_pc
B.U (64) = pw_useru
B.UC (63) = pw_ucomm
B.KFTP=0
T.TRACEM=tracemask
any xp flags set (bits in S1)
trapusr[] entry:
handler address
trace ascii
signal
B.SUBR (04) = trapusr[flag]
S1 = trapusr[] trace info / signo
/usr/src/lib/asdef/asdef.s
B.%STKCBP (02) = &u_stack
B.%STKCTP (66) = frame[1]
B.%STKATP (67) = limit
"Current Base"
"Current Top"
"Absolute Top"
B.SUBR
Set IOI in S1
(else clear IOI in XP)
Set ICP in S1
Clear ST7 bit
Set PCI in S1
pcilock = 1
this cpu's bit set in ST7
!pcilock && RTC > nextpci
usrnex
S3 = user S0 S4 = user S1
p_pflag & P_VM
vmtrap
copy the XP to the VM process
to usrnex02
no arguments
validate user arg list
call is interruptable (36 of them are)
B00=stjmp1
quick stack context save
in u_save[U_QSAV]
(B00, B01, B02, B066, B067, T00)
stjmp2:
member of multitasked group AND call is not multi–threaded (22 of them are)
mtasklock
if PFROZE call qswtch
(longjmp on sleep interrupted by a signal returnes here)
stjmp1:
if u_error = 0
u_error=EINTR
p_intproc != 0
call processing
to usrnex01
p_bsctime = RTC
p_syscall = call #
B00 = usrnex01
205 system call handlers
sleep
interruptable and processable signal
jump B00
usrnex01
member of multitasked group AND call is not multi–threaded
mtaskunlock
clear pc_mtask
user S1 = RVAL1
user S2 = RVAL2
user S0 = u_error
if over 20 errors this process (u_errcnt) set pw_users
usrnex02
calculate statistics (if SCTRACE)
p_syscall = 0
shconsts.sc_csyscall ++
p_pflag & P_VM
preserve XP flags
to master
usrioi
set IOLOCK (TSF$IO)
ST.IOFLAG
ST.IOFLAG = nonzero
usrioi00
get shared registers (ST.CHANF,ST.CHANL,ST.CHANR)
usrioi01
CI!=0
channel in ST.CHANL
mark channel in ST.CHANR
usrioi03
test ST.CHANR (S7) | ST.CHANF (S2)
none pending
wait until done
channel # = Z(ST.CHANR|ST.CHANF)
pseudo
clear bit ST.CHANR
clear bit ST.CHANF
SSD
clear chan_acct
clear from ssd_actmask
others active
clear IOLOCK
(&chan entry, 1)
VHISP
LOSP (even)
LOSP
LOSP (custom)or
TCP pseudo
soft_timeout
set IOLOCK (TSF$IO)
usrioi04
ST.CHANL
trace "IOM"
CI!=0
trace "IOR"
usrioi05
ST.IOFLAG = 0
ESI (C90)
usrioi06
clear IOLOCK
usrfpi
usrore
usrpre
usrbpi (C90)
usreex
usrerex
(each entry point sets an offset to its error type count field in the sysinfo structure)
trace ("FPI" / "ORI" / "PRE" / "BPI" / "EEX")
sysinfo.usrex[cpu]
(debugging and statistics)
IPI any other cpu's in this MT group (and set its pw_users)
(all members of the group need to check their signals if this is a fatal signal – the first to die will SIGKILL the others)
usrsig
psignal
signal the process with signal from trapusr[]
usrmii
PANICNE ("master.s: MII interrupt from userland")



usrmei

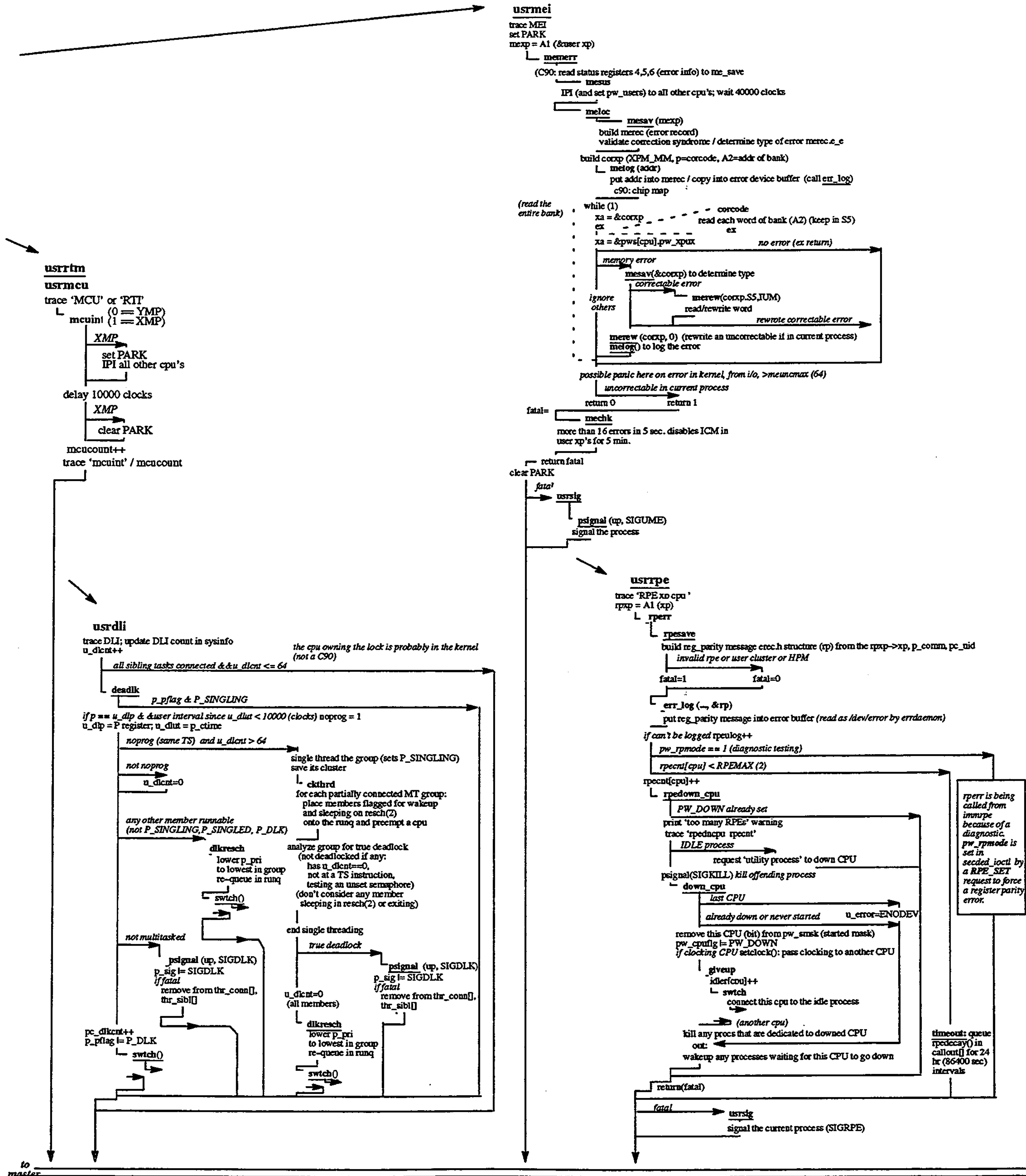
trace MEI
set PARK
mexp = A1 (&user xp)
memerr
(C90: read status registers 4,5,6 (error info) to me_save
mesus
IPI (and set pw_users) to all other cpu's; wait 40000 clocks
meloc
mesav (mexp)
build merec (error record)
validate correction syndrome / determine type of error merec.e_e
build corxp (XPM_MM, p=corcode, A2=addr of bank)
melog (addr)
put addr into merec / copy into error device buffer (call err_log)
c90: chip map
(read the entire bank)
while (1)
xa = &corxp
ex
corcode
read each word of bank (A2) (keep in S5)
ex
xa = &pws[cpu].pw_xpux
no error (ex return)
memory error
mesav(&corxp) to determine type
correctable error
ignore others
merew(corxp,S5,IUM)
read/rewrite word
rewrote correctable error
merew (corxp, 0) (rewrite an uncorrectable if in current process)
melog() to log the error
possible panic here on error in kernel, from i/o, >meuncmax (64)
uncorrectable in current process
return 0 return 1
fatal=
mechk
more than 16 errors in 5 sec. disables ICM in user xp's for 5 min.
return fatal
clear PARK
fatal
usrsig
psignal (up, SIGUME)
signal the process

usrrpe

trace 'RPE xp cpu'
rpxp = A1 (xp)
rperr
rpesave
build reg_parity message erec.h structure (rp) from the rpxp->xp, p_comm, pc_uid
invalid rpe or user cluster or HPM
fatal=1 fatal=0
err_log (..., &rp)
put reg_parity message into error buffer *(read as /dev/error by errdaemon)*
if can't be logged rpeulog++
pw_rpmode == 1 (diagnostic testing)
rpecnt[cpu] < RPEMAX (2)
rpecnt[cpu]++
rpedown_cpu
PW_DOWN already set
print 'too many RPEs' warning
trace 'rpedncpu rpecnt'
IDLE process
request 'utility process' to down CPU
psignal(SIGKILL) *kill offending process*
down_cpu
last CPU
already down or never started u_error=ENODEV
remove this CPU (bit) from pw_smsk (started mask)
pw_cpuflg |= PW_DOWN
if clocking CPU setclock(): pass clocking to another CPU
giveup
idler[cpu]++
swtch
connect this cpu to the idle process
(another cpu)
kill any procs that are dedicated to downed CPU
out:
wakeup any processes waiting for this CPU to go down
return(fatal)
fatal
usrsig
signal the current process (SIGRPE)

rperr is being called from immrpe because of a diagnostic. pw_rpmode is set in secded_ioctl by a RPE_SET request to force a register parity error.

timeout: queue rpedecay() in callout[] for 24 hr (86400 sec) intervals



to master

IDLE process

slaveidl

down CPU

down CPU: idle or diagnostic

pw_users

qswtch
disconnect old process
select new from runq
connect new (longjmp)

xp base=0,
xp limit=physmem
xp P=idleloop
xa=pw_xpus
ex

idleloop
(monitor wakeup flags)

EMI
xa=pw_xpux
pw_idlet += (RT – pw_rtsav)
p_stime += (RT – pw_rtsav)

Y–MP or C90 and RPE
save RPE info
TRACE "IPR"

nex (idleloop has detected a wakeup)

ckthrd
for each partially–connected MT group:
place disconnected,
wakeup–flagged members
sleeping in resch(2) onto runq

ilkthr = 0 *(set in idle)*
TRACE "IWK"

to master

slaveusr

pw_users or
pc_flag || PC_SSPND, PC_FREEZE, PC_SINGL
or
signal pending (p_sig) or
p_flag || P_PRSTOP

B.%STKCBP (02) = &u_stack
B.%STKCTP (66) = frame[1]
B.%STKATP (67) = limit

pw_users or
pc_flag || PC_SSPND, PC_FREEZE, PC_SINGL
pw_users = 0 *(time to check the runq)*

qswtch p_pri=p_upri
p_slp_action=kstack_mpx_noaction

swtch
Disconnect current proc
and save resume point
(with setjmp) *(swtch() + nnn)*

(longjmp)

sureg
Finish XP
and pws

Find first qualified proc on the runq

resume
Swap XP's, swap clusters, connect new proc
Set global pointers B.UP B.U B.UC B.UPC

longjmp to new proc's resume point
(loads B and T registers from u_save)

B00

signal pending (p_sig) or
p_flag || P_PRSTOP

!PRSTOP

issig
Find any signal in
(p_sig) that is not to
be ignored, held or off

sendable signal found
or PRSTOP

psig

PRSTOP
send SIGSTOP to all MT members
swtch: disconnect process
clear SIGSTOP in p_sig

sendable signal found

signal not registered
to be caught

save user XP at p_sigoff–>sigarea
P = handler (u_sigact[])
A4 = signal number
A5 = sigon/sigoff
clear p_sig

SIGKILL all MT members
if p_sigdmp
core:write core file

zexit
deallocate
resources

swtch

if BMM and u_bmmsaved
restore BMM unit from u_savebmm

NEX

restore V0
restore V1 or V1–7
restore VM/VM1

V0 = 0
V1 or V1–7 = 0

A1 = &pw_xpus

trace 'USR'
A2 = up
A3 = uc
A5 = u
if VM restore XP
flags
restore B1–63
restore T0–63

from SSD hot path

CALLHOT
S4 = RTC
p_stime += (RTC – pw_rtsav)
pw_unixt += (RTC – pw_rtsav)
pw_rtsav = RTC
S5 = p_ctime.
(user) pw_utms–>utms_update = RTC
(user) pw_utms–>utms_utime = p_ctime
(from utimes() call)

p_pri = p_upri
pw_cpri = p_upri

update uc and pw_mtms–>mtms area
(from mtimes() call)

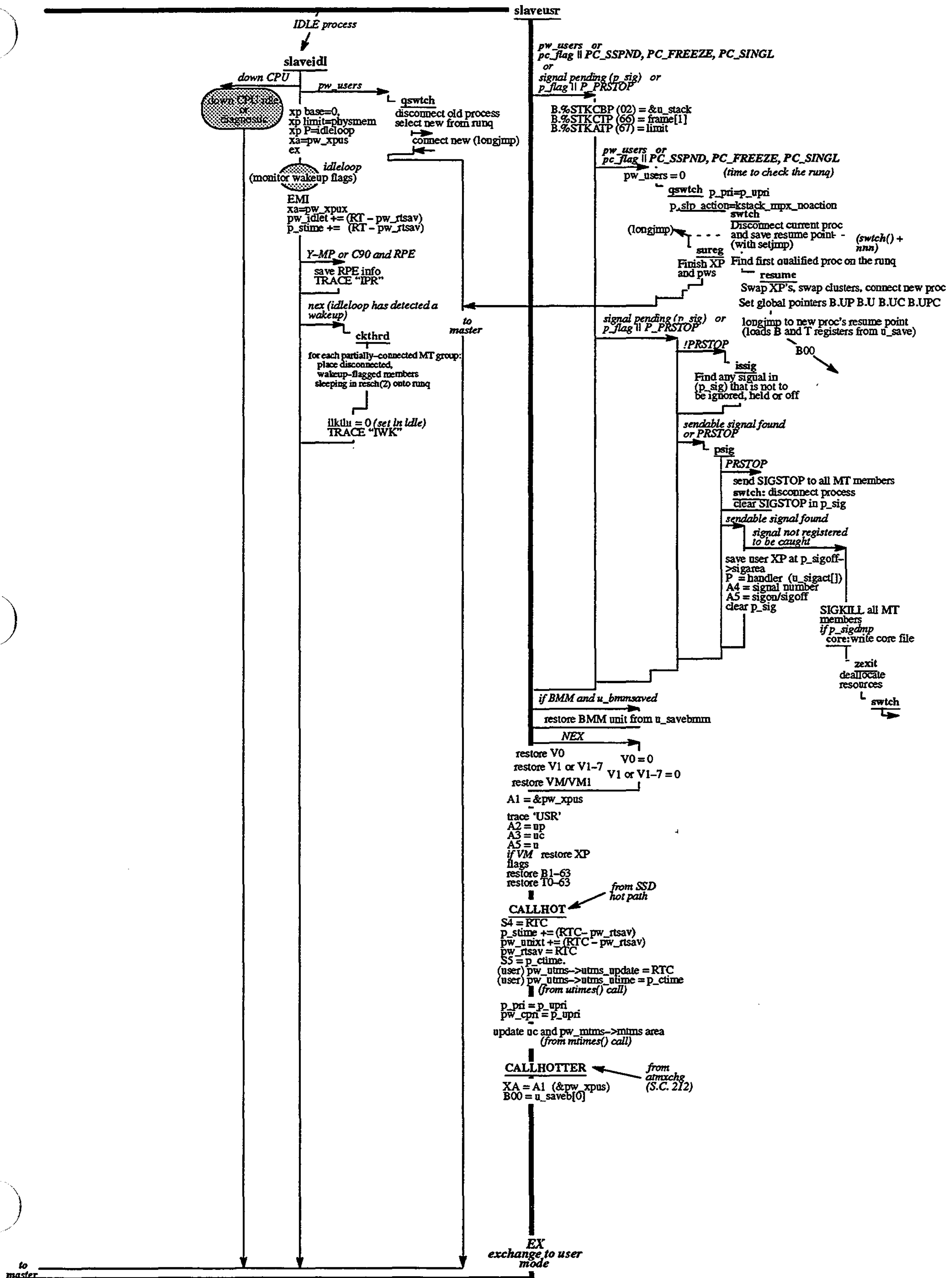
CALLHOTTER

from atmxchg (S.C. 212)

XA = A1 (&pw_xpus)
B00 = u_saveb[0]

EX
exchange to user mode

to master



slaveusr
IDLE process
slaveidl
down CPU
down CPU, idle or diagnostic
pw_users
qswtch
disconnect old process
select new from runq
connect new (longjmp)
xp base=0,
xp limit=physmem
xp P=idleloop
xa=pw_xpus
ex
idleloop
(monitor wakeup flags)
EMI
xa=pw_xpux
pw_idlet += (RT – pw_rtsav)
p_stime += (RT – pw_rtsav)
Y-MP or C90 and RPE
save RPE info
TRACE "IPR"
nex (idleloop has detected a wakeup)
to master
ckthrd
for each partially-connected MT group:
place disconnected,
wakeup-flagged members
sleeping in resch(2) onto runq
ilktlu = 0 (set in idle)
TRACE "IWK"
pw_users or
pc_flag || PC_SSPND, PC_FREEZE, PC_SINGL
or
signal pending (p_sig) or
p_flag || P_PRSTOP
B.%STKCBP (02) = &u_stack
B.%STKCTP (66) = frame[1]
B.%STKATP (67) = limit
pw_users or
pc_flag || PC_SSPND, PC_FREEZE, PC_SINGL
pw_users = 0
(time to check the runq)
qswtch
p_pri=p_upri
p_slp_action=kstack_mpx_noaction
swtch
Disconnect current proc
and save resume point
(with setjmp)
(longjmp)
(swtch() + nnn)
sureg
Finish XP
and pws
Find first qualified proc on the runq
resume
Swap XP's, swap clusters, connect new proc
Set global pointers B.UP B.U B.UC B.UPC
longjmp to new proc's resume point
(loads B and T registers from u_save)
B00
signal pending (p_sig) or
p_flag || P_PRSTOP
!PRSTOP
issig
Find any signal in
(p_sig) that is not to
be ignored, held or off
sendable signal found
or PRSTOP
psig
PRSTOP
send SIGSTOP to all MT members
swtch: disconnect process
clear SIGSTOP in p_sig
sendable signal found
signal not registered
to be caught
save user XP at p_sigoff->sigarea
P = handler (u_sigact[])
A4 = signal number
A5 = sigon/sigoff
clear p_sig
SIGKILL all MT members
if p_sigdmp
core: write core file
zexit
deallocate resources
swtch
if BMM and u_bmmsaved
restore BMM unit from u_savebmm
NEX
restore V0
restore V1 or V1–7
restore VM/VM1
V0 = 0
V1 or V1–7 = 0
A1 = &pw_xpus
trace 'USR'
A2 = up
A3 = uc
A5 = u
if VM restore XP
flags
restore B1–63
restore T0–63
from SSD hot path
CALLHOT
S4 = RTC
p_stime += (RTC– pw_rtsav)
pw_unixt += (RTC – pw_rtsav)
pw_rtsav = RTC
S5 = p_ctime.
(user) pw_utms–>utms_update = RTC
(user) pw_utms–>utms_utime = p_ctime
(from utimes() call)
p_pri = p_upri
pw_cpri = p_upri
update uc and pw_mtms–>mtms area
(from mtimes() call)
CALLHOTTER
from atmxchg (S.C. 212)
XA = A1 (&pw_xpus)
B00 = u_saveb[0]
EX
exchange to user mode
to master

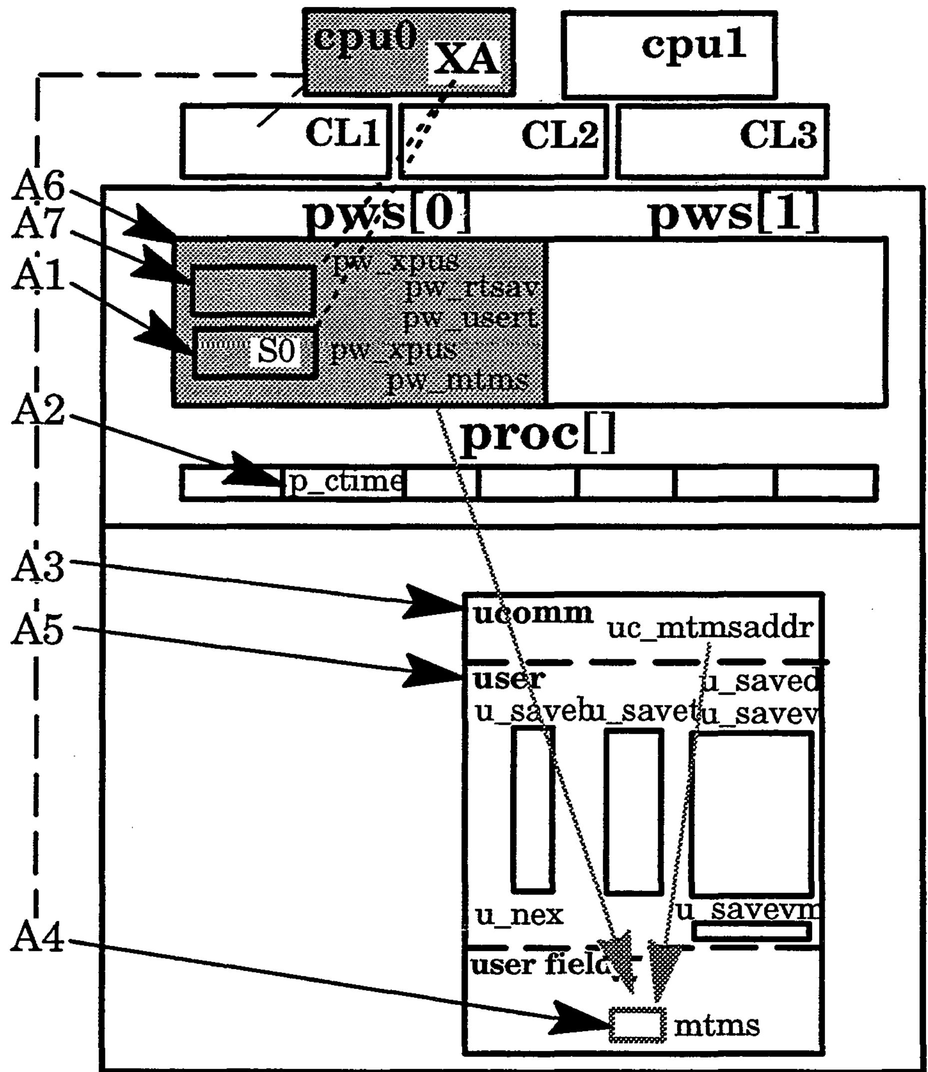
Mainline outer loop

Kernel entry

The diagram on the right shows the main points of interest in the logic flow of the "entry housekeeping". Pictured below are the data structures involved.

Registers at entry

A CPU leaves the kernel to any user process, and reenters again, around line 460 of `/usr/src/uts/c1/md/slave.s`. The A and S registers are identical on exit and reentry from/to the kernel because they are saved and restored by the hardware exchange sequence. Just before the CPU exchanges to user mode the A registers point to the connected process' main structures



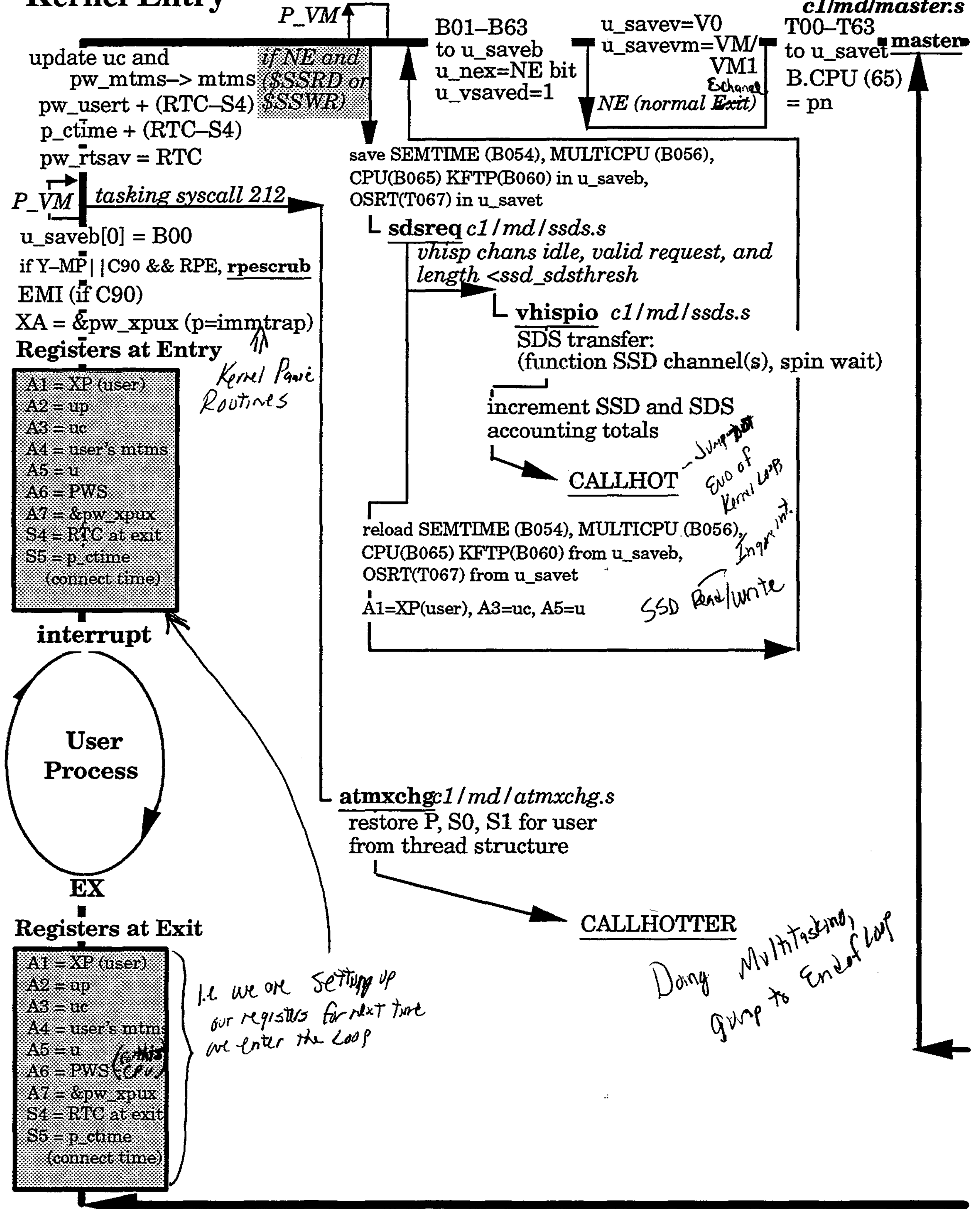
Initial saves and updates

`XA` needs to point to the error-handling exchange package `pw_xpux` while the CPU is in kernel mode in case of a hardware memory error (or other error). `EMI` is a C90 instruction to enable the interrupts specified in the kernel's interrupt modes register (namely RPE, IUM, IFP, IOR, IPR). These would not otherwise cause interrupts to a C90 CPU in monitor mode. A non-C90 is in IMM mode without this instruction. See the section "immtrap".

A Register Parity error (RPE) is dealt with specially. Function `rpescrub` is called to "scrub" the user's B, T, and V registers by saving and reloading them. (See the sections "`immtrap`" and "`usrrpe`" later in this chapter.) B00 is saved specially because it is used by hardware for any subroutine call.

(—text continued after the next page—)

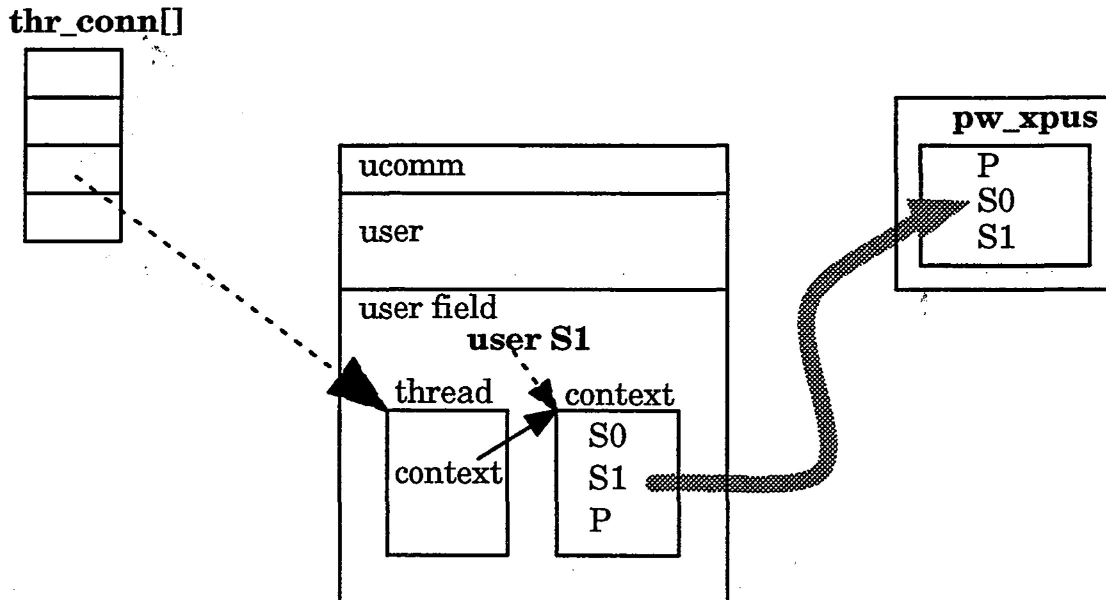
Kernel Entry



Initial saves and updates: (continued)

Autotasking system call 212: "atmxchg" stands for "auto-tasking mini-exchange". This call ($$MXCHG) restores a user's P, S0 and S1 registers from its context to its exchange package, and sets the context pointer. The library routine wishes to load up the entire register context of an interrupted task, but cannot load all registers and then jump to some entry point without resorting to self-modifying code. And self-modifying code is impossible if the program is text that can be shared. So the library routine loads all registers except the P register and the two S registers needed to make the call, and the kernel restores the P, S0 and S1. This call also closes a "window" of time in dealing with the thread pointer to context when the hardware could interrupt a process after it marks its thread to save in the context but before the process finishes loading up the registers already in that context structure. (See the "Cooperative Parallel Interface" section in chapter 5 for more details.)

Note: This work is accounted as user time. The kernel exits immediately at CALLHOTTER, doing no system time accounting. ♦



"RTC" is the real-time clock. It is saved immediately in pw_rtsav for accounting purposes. The difference between the RTC "now" and at last exit from the kernel is "user time". This is added to the connected process' (p_ctime) and CPU's (pw_usert).

Updating of the process' "mtms" is only valid for a process that has made a multitasking mtimes(2) call. If not, these pointers point to a kernel "scratch" area.

$SSRD/$SSWR: A special case occurs when a process makes an `ssread(2)` or `sswrite(2)` system call (a transfer bewteen a user buffer and SSD Secondary Data Segments – (see the discussion of SDS in the I/O chapter). SSD VHISP channels are so fast that the save/restore of user register context is a relatively unreasonable amount of overhead.

If the channel/s is/are not busy, the transfer parameters are valid, and the transfer size is below an SDS transfer limit (set in `sdsreq`), this driver is called to perform the fast transfer and jump to the CALLHOT exit back to user mode. This path is called the "hot path". If the above test fails or there is an I/O error the system call is performed in the same manner as any others.

Actual hotpath RTC clock times on: `sys:sn4809    node:wind    rel:8.0.2 ver:mpr.106`

(**Note:** The 12000 block case was above **ssd_sdsthresh** hot path limit of 10000 blocks.)♦

5000 **ssread()** calls (10 times for 12000 case)

Blocks transferred	Best time	Best (per block)	Worst time	Average time
1	2614	2614	31703	2652
10	4613	461	37677	4721
50	14859	297	83964	15051
100	27694	276	269812	28618
12000	3118696	259	4656142	3136935

5000 **sswrite()** calls

Blocks transferred	Best time	Best (per block)	Worst time	Average time
1	2614	2614	95123	2660
10	4560	456	5282	4624
50	14808	296	208366	15062
100	27621	276	963642	30928
12000	3117534	259	3151426	3124337

Compare to 5000 **getpid()** calls:

Release	Best time	Worst time	Average time
7.0.?	2866	194956	5060
8.0.2	2643	111534	2812

The more blocks transferred with a single I/O request, the better (as would be expected). Worst times and average times are not very significant as they can reflect times when the hot path or semaphore locks were busy, and therefore may vary greatly depending on the load of the system.

Save of B's, T's and V's: The B and T registers (0–63) are going to be used extensively by the kernel's C code, so they are preserved unconditionally. The kernel is compiled in non-vectorizing mode, however, so vector registers are only used by CAL routines.

Only V0 is normally used (by bwcopy, bzero), so it and the vector mask (VM) are saved. (VL, vector length, is saved in the xp during the exchange.) The VM is 128 bits (VM/VM1) on a C90.

`u_vsaved` is a count of vectors saved. It can contain 1, 2 or 8. It is set to 1 here.

If a CAL routine such `bcopy` or `strlen` uses V1 it first saves V1 and changes `u_vsaved` to 2. No other vectors need be saved until the process is disconnected, at which time `u_vsaved` is set to 8.

Vectors are restored according to the `u_saved` count at exit from the kernel.

Exception: Vector registers are not preserved across a system call. They are set to 0 on return to the user, which is much faster. No CRI compilers vectorize across a function call, so this is no problem from compiler-generated code. Anyone coding in CAL should be aware of this fact. ♦

This page used for alignment

immtrap - *Trapping monitor mode interrupts* — Kernel Panic Routines

Exchange with 0: As a precaution against a hardware error causing an exchange with zero and subsequent unpredictable execution, location 0 is initialized with an exchange package whose p register will take such a CPU to the panic() call at zerotrap().

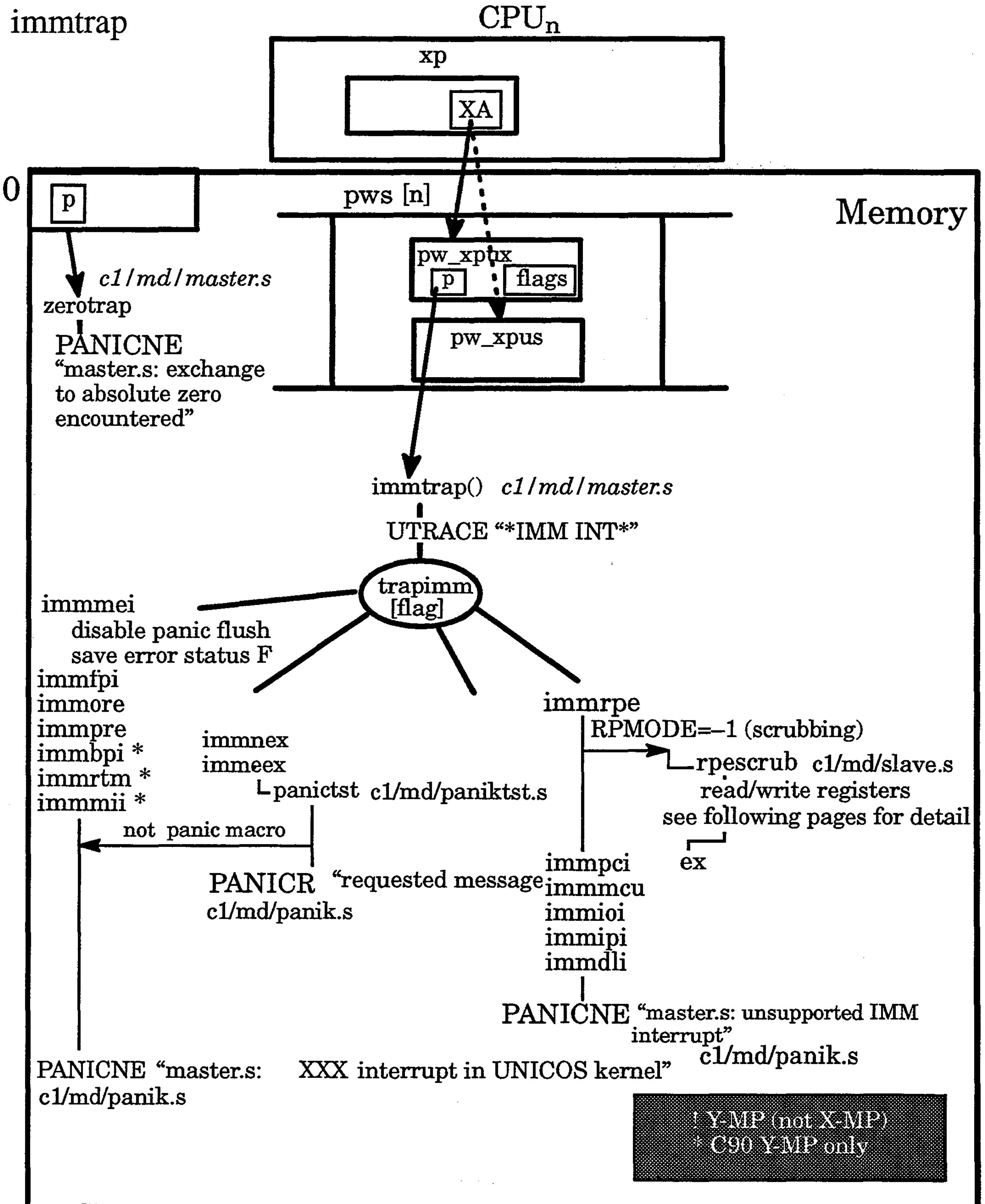
Panic: panic() (sysmacros.h) and PANIC (utext.s) are macros that cause a normal exit or error exit in the kernel. Therefore all system panics execute immnex or immeex. The macros generate "calling sequences" which panictst() can identify. The PANICNE macro used here jumps to the PANICR entry point (in md/panik.s) that hangs the machine.

immtrap routines: The diagram shows what happens if the UNICOS kernel is interrupted while executing in monitor mode MM and the XA is set to pw_xpux (p=immtrap). Recall that on entry to the kernel the hardware exchange address (XA) register is loaded with the address of the CPU's pw_xpux (unix) exchange package. Just before exchanging back to a user process the XA is reset to the pw_xpus (user) exchange package.

CRAY Y-MP or X-MP: The UNICOS kernel executes with Monitor Mode (MM) and Interruptible Monitor Mode (IMM) mode bits set.

CRAY Y-MP C90: The UNICOS kernel executes in Monitor Mode (MM); there is no IMM mode. The following interrupts are enabled in the C90 kernel: ORE, FPE, MEU,RPE, PRE, NEI, EEI. (NEI (and EEI/PRE) interrupts are never maskable, NEI flag is masked off. An exchange with no flag is assumed to be an NEI.) MEC and MEU are both handled by immmei.

routine	interrupt	meaning	action
immmei	Memory error IUM / ICM	Double bit error or single bit error caused interrupt (singles are disabled) – hardware error	Save status disable panic flush PANIC "MEI"
immfpi	Floating point FPI	Kernel logic error –OR– hardware error	PANIC "FPI"
immore	Operand range error ORE	Kernel logic error –OR– hardware error	PANIC "ORE"
immpre	Program range error PRE	Kernel logic error –OR– hardware error	PANIC "PRE"
immbpi Δ	Breakpoint interrupt	Write reference to breakpoint address should not happen in MM – hardware error	PANIC "BPI"
immrtm Δ	Real time RTM	Unsupported hardware – interrupt indicates hardware problem	PANIC "RTM"
immmii Δ	Monitor mode interrupt MII	Interrupt occured for executing a MM instruction – since kernel in MM this indicates a hardware problem	PANIC "NEX"
immnex	Normal exchange NEX	Accidental" execution of an EX command (hardware or software error) or EX as result of panic macro	If by C panic PANIC "user message" else PANIC "NEX"
immeex	Error exchange EEX	"Accidental" execution of an EE command (hardware or software error) or EE as result of panic macro	If by CAL panic PANIC "user message" else PANIC "EEX"
immrpe †	Register parity error RPE	Kernel encounters parity error in CPU registers – hardware error	If "scrubbing" read/write registers else PANIC "RPE"
immmcu	Maint. control unit interrupt	Should be held pending in MM – hardware error	PANIC "MCU"
immioi	I/O interrupt	Should be held pending in MM – hardware error	PANIC "IOI"
immipi	Interprocessor interrupt	Should be held pending in MM – hardware error	PANIC "IPI"
immdli	Deadlock interrupt	All CPU in kernel cluster 1 holding on a TS command – logic –OR– hardware error	PANIC "DLI"
immpci	Programmable clock interrupt	Should be held pending in MM – hardware error	PANIC "PCI"


immtrap
CPUn
xp
XA
0
p
pws [n]
Memory
pw_xpux
p
flags
pw_xpus
c1/md/master.s
zerotrap
PANICNE
"master.s: exchange
to absolute zero
encountered"
immtrap() c1/md/master.s
UTRACE "*IMM INT*"
trapimm
[flag]
immmei
disable panic flush
save error status F
immfpi
immore
immpre
immbpi *
immrtm *
immmii *
immnex
immeex
panictst c1/md/paniktst.s
not panic macro
PANICR "requested message
c1/md/panik.s
immrpe
RPMODE=–1 (scrubbing)
rpescrub c1/md/slave.s
read/write registers
see following pages for detail
ex
immpci
immmcu
immioi
immipi
immdli
PANICNE "master.s: unsupported IMM
interrupt"
c1/md/panik.s
PANICNE "master.s: XXX interrupt in UNICOS kernel"
c1/md/panik.s
† Y-MP (not X-MP)
* C90 Y-MP only

`rpescrub` *and* `immrpe`

Exchange with 0: As a precaution against a hardware error causing an exchange with zero and subsequent unpredictable execution, location 0 is initialized with an exchange package whose p register will take such a CPU to the `panic()` call at **`zerotrap()`**.

`rpescrub`: The code to call `rpescrub` is generated on CRAY Y-MP and CRAY Y-MP C90 systems only. In the event of a register parity error from the "user" RPE exchange, this routine is called to "scrub" the hardware registers "attempting to" prevent additional RPE interrupts within the kernel as the kernel processes these register values for the user (for example, saves them).

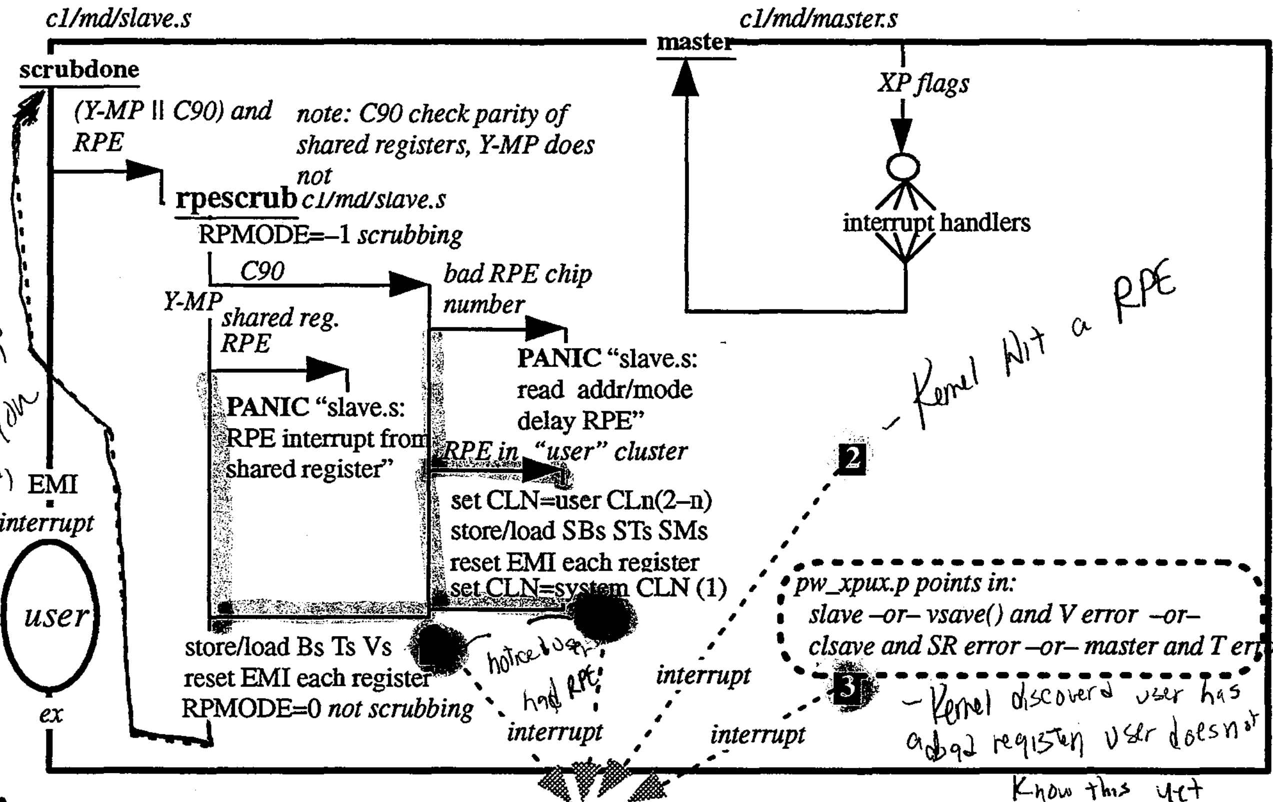
The code sets flag `RPMODE` indicating that scrubbing is taking place. This flag protects the kernel from the normal panic action that would occur for an RPE within the kernel, as described in `immrpe`. The logic simply saves each register in memory and then loads the value back in the register. The parity of the value should be "fixed" by this action provided that the error is intermittent. A solid RPE will panic or hang the system in a very short time. Shared registers in the user's assigned cluster are scrubbed on C90 systems only. The `RPEMODE` flag is cleared when the scrubbing is complete. Also note for C90 systems the `EMI` instruction is executed after each register is scrubbed since any interrupt exchange would have implicitly turned this mode off.

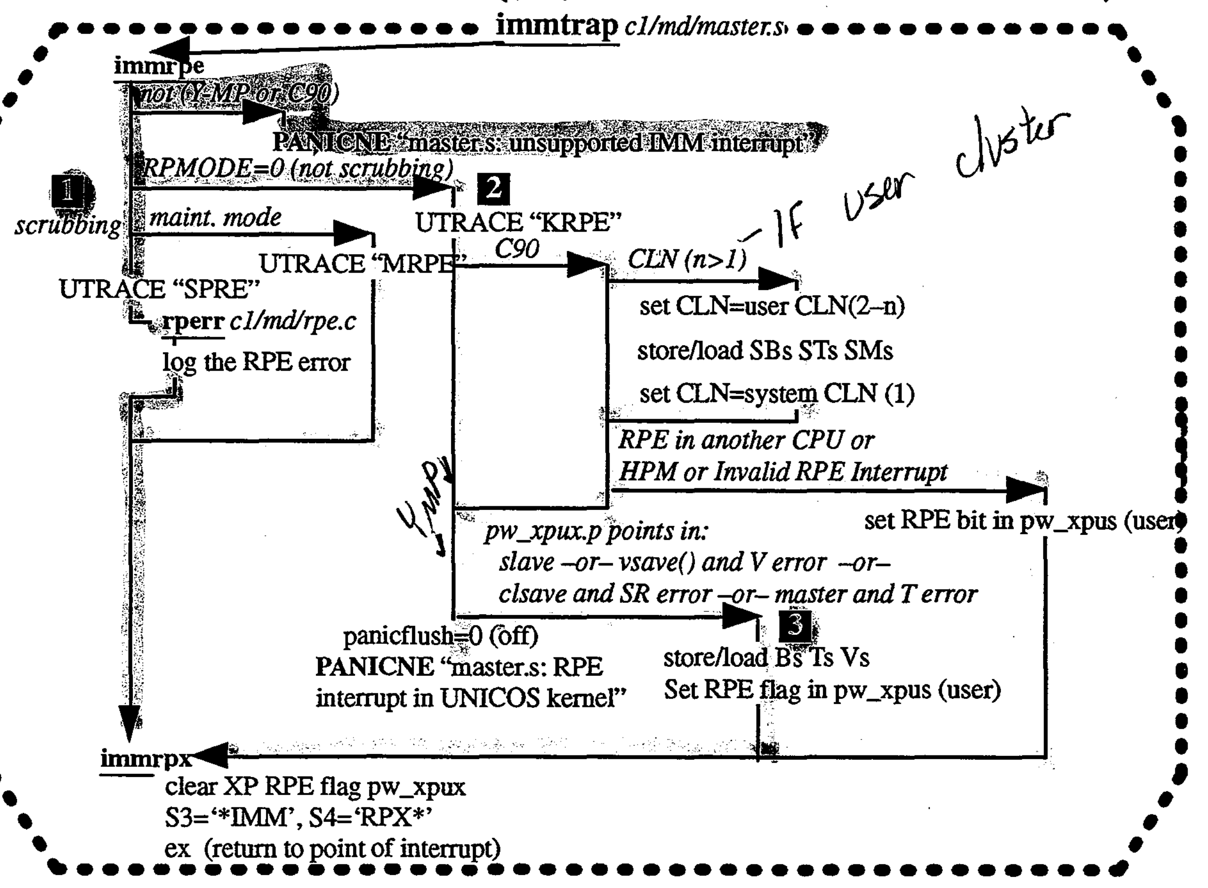
`immrpe`: CPU enter `immrpe` through `immtrap` for any register parity error interrupt exchanges. The interrupt could occur in 3 major contextual areas in the kernel as shown on the right and described below:

1. During register scrubbing (`RPMODE` set) – `immrpe` checks if the cluster (user process) is in maintenance mode, a privileged function set in `/dev/cpu` for testing shared registers. In maintenance mode the error is ignored here, it will be processed by `usrrpe` later in the logic. In "normal" operation the error is logged by `rperr`. In any case the CPU is exchanged "back" to the instruction after the load/save that caused the error and processing continues.
2. During register use by the kernel (with exceptions listed in number 3) – `panicflush` is turned off and the system panics with the RPE message.
3. During register use by the kernel but:
   - RPE was in another CPU, in HPM, or there was an invalid RPE interrupt – the RPE bit is set in the user exchange package forcing the error to be processed by `usrrpe`.
   - Within the routines that are processing "user" register data, not kernel data: `slave.s`, `vsave()`, `clsave()`, or `master.s` and T register reference. In these cases the kernel mimics the logic of `rpescrub` again here. The system is protected from panicking in this case. Also the RPE bit is set in the user exchange package forcing the error to be processed by `usrrpe`.

In all cases except the kernel RPE panic, the CPU is exchanged "back" to the instruction after the load/save that caused the error and processing continues.

rpescrub and immrpe





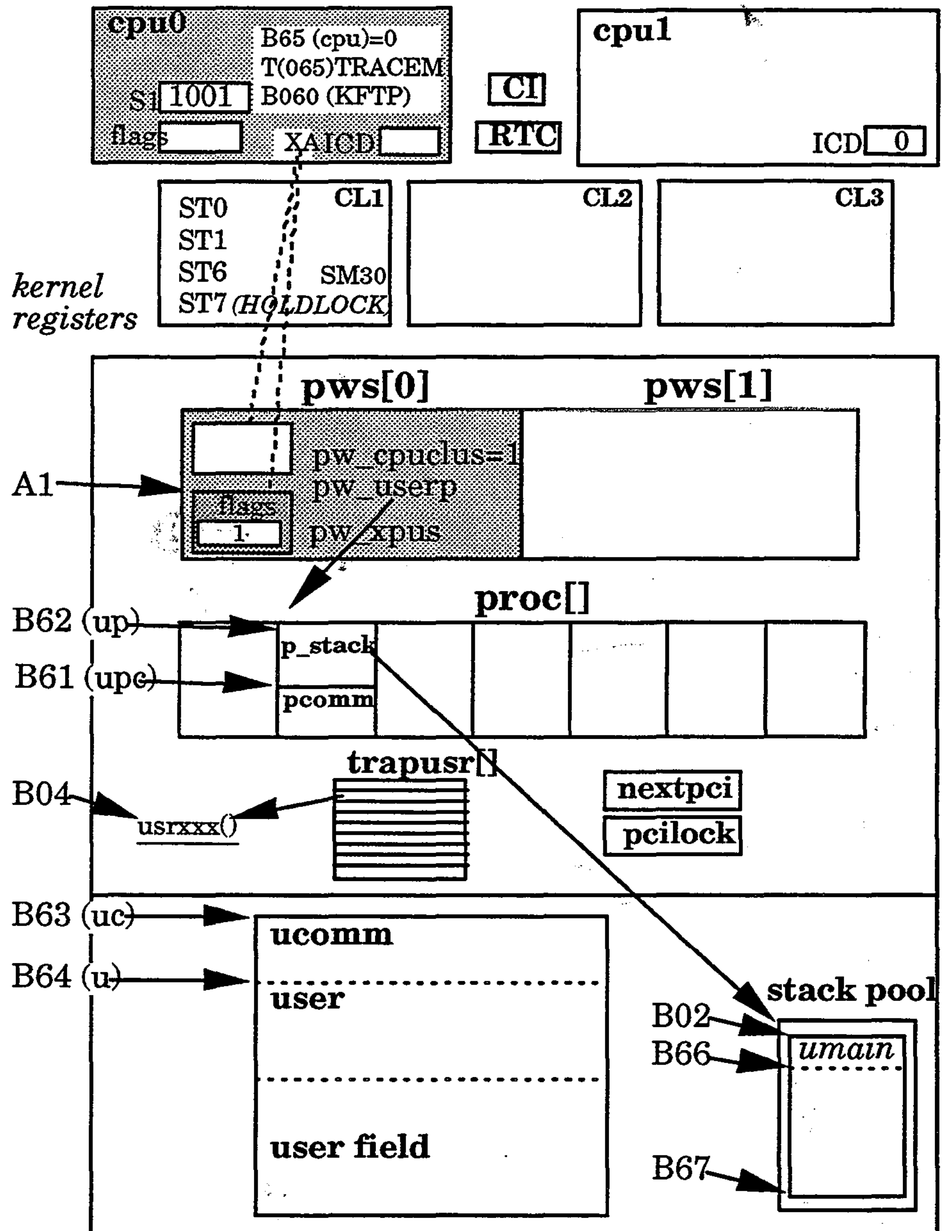
Interrupt handler selection

The diagram on the right shows the logic of selecting the next interrupt to handle in the “inner loop”. Pictured below are the data structures involved.

Recall that **pw_xpus** contains the user exchange package saved when the interrupt occurred. The XP flags tell us why the CPU entered the kernel.

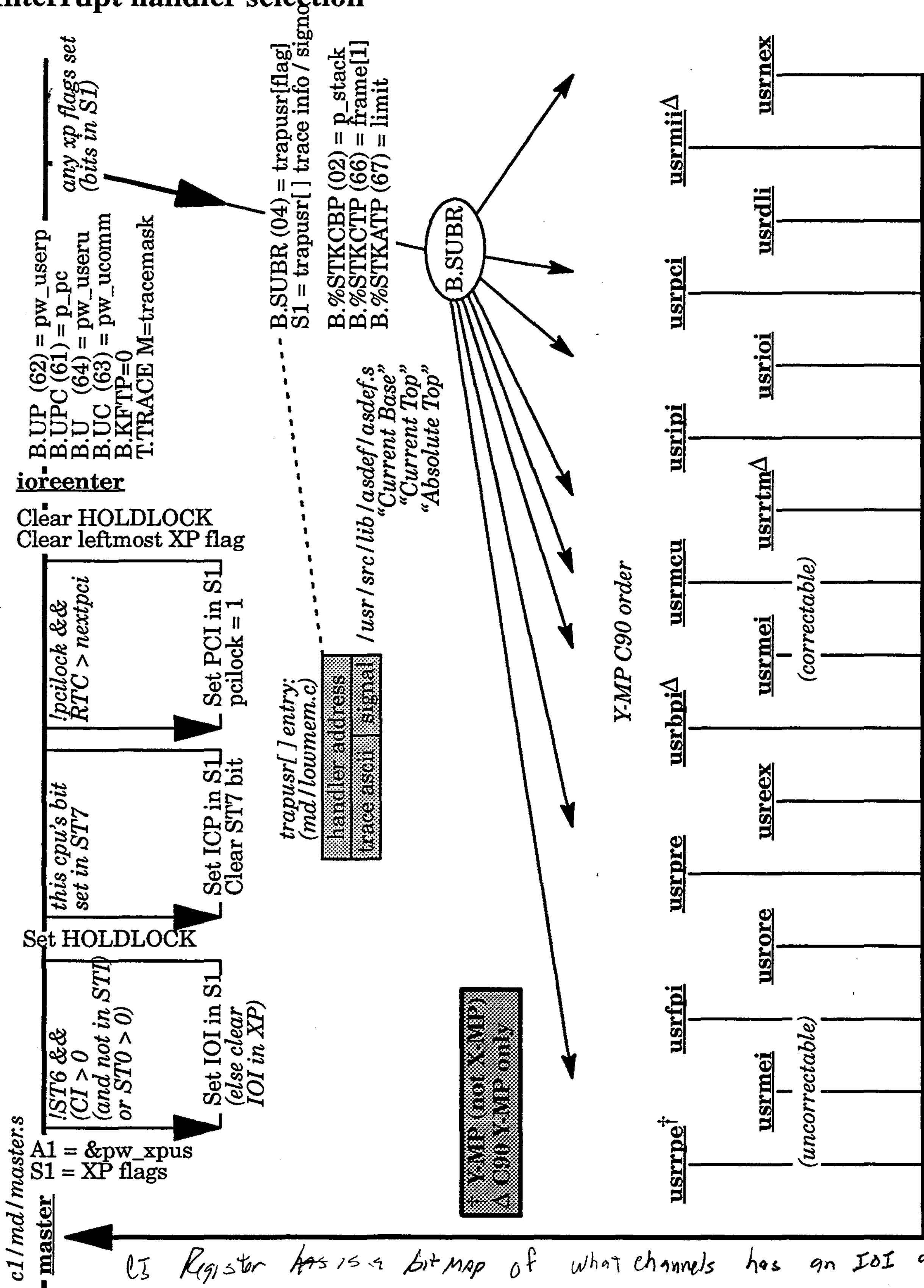
Generally there is only one flag bit on in the user exchange package set as the result of the hardware interrupt but there are 3 “software interrupts” or “pseudo interrupts” checked here. This logic detects any possible “pending” IOI, PCI, and IPI interrupts. By detecting these here with software checks the CPU is saved from trying to exchange back to the user only to exchange right back into the kernel to process the interrupt. Each test possibly sets the corresponding “real” interrupt flag in the memory XP (pw_xpus) resulting in the interrupt being processed as if it were caused by a “real” hardware interrupt.

The kernel forms the XP flags into a single word and then simply does a leading zero count of the flag word to determine what is the leftmost flag, and uses that number as an index into the `trapusr[]` jump table.



Each interrupt handler clears its corresponding XP flag bit and normally jumps to **master** when it completes its activity. The CPU loops through this logic until the last XP bit is cleared.

Interrupt handler selection



IOI: (CRAY X-MP/CRAY Y-MP) An I/O interrupt can occur at any time. If channel-completion actually caused the exchange in this CPU, the IOI bit is already set. But if a channel completes a transfer while this CPU is already in monitor mode, the channel will direct the interrupt to this CPU (refer to the IOI bit's description in the hardware chapter). The I/O interrupt remains pending for this CPU until it either returns to user mode or clears the pending interrupt via a machine instruction.

IOI: (CRAY-MP **C90**) An I/O interrupt can occur only when the SIE hardware "gate" or "flag" is set, which is when all CPUs are in user mode, or when a CPU has entered the kernel, handled all pending I/O interrupts, and executes the ESI instruction shortly before going back to user mode. The kernel executes with I/O interrupts disabled.

CI: The CPU should stay in the kernel and "poll" the CI (Channel Interrupt) register for any pending interrupt. Refer to the hardware section for more information.

ST6: I/O interrupt lockout. Refer to the `usrioi` routine to see that the ST6 register (in cluster 1) is nonzero when any CPU is already in the I/O interrupt handler. That CPU will handle all pending I/O interrupts.

ST1:	Lockout channels	Count of CPU's in usrioi interrupt handler

The left side of ST1 is a bit map of which (SSD) channels are currently being polled by a CPU. A CPU which functions one or more SSD channel/s and then spin-waits for the transfer's completion will set those channel's bits in ST1. A CPU entering the kernel should ignore those channel/s.

ST0: Bit map of pseudo channel interrupts.

TCP/IP postpones full processing of a message until all real channels have been handled. It does this by flagging a bit in ST0 representing a channel number above the possible real channel numbers. Logic also supports pseudo LOWSP and VHISP channels for special purposes such as memory to memory (system to system) communication in a UNICOS guest environment. See the `usrioi` routine.

HOLDLOCK: A hardware semaphore (SM30) used to protect against simultaneous update of ST7 or `pcilock`.

`ICP`: The Internal Central Processor flag is also known as an "Interprocessor Interrupt" or "**IPI**" (it is referred to as "IPI" in the Hardware section).

Any CPU in monitor mode can raise this interrupt in any other CPU by means of a machine instruction (SIPI). The exception to this rule, and therefore the reason for this software test, is that the SIPI instruction will not interrupt a CPU in monitor mode. (On a **C90**, the kernel runs without the IPI enabled.) The interrupt would remain pending until the CPU goes back to user mode.

ST7: A bit map of software IPI's.

To notify a CPU that is already in monitor mode that it should execute the `usripi` interrupt handler, the "sending" CPU does a SIPI to the "target" CPU and sets the target's bit (counting from the left) in ST7. See the IPI macro in `md/utext.s`.

PCI: A Programmable clock interrupt can occur at any time. If the programmable clock actually caused the exchange in this CPU, the PCI bit is already set. But if its clock (ICD) counts down to 0 while this CPU is already in monitor mode, the interrupt remains pending until this CPU either returns to user mode or clears the pending interrupt via a machine instruction. (In a **C90** the kernel runs without PCI enabled.) Normally only 1 CPU, "the clocking CPU", is running its programmable clock.

See the Hardware chapter for more description of the programmable clock in each CPU.

> `nextpci`: The real time at which the next timed event should be performed is stored in nextpci. The clocking CPU can use this to detect the pending interrupt without going back to user mode. Or any other CPU can perform the scheduled service if the clocking CPU is "tied up" elsewhere in the kernel.
>
> `pcilock`: Keeps the `pcintrpt()` routine single-threaded. It is nonzero when a CPU is currently performing the scheduled service. (See the `usrpci` interrupt handler.)

Global pointers: B61, B62, B63, B64 and B65 are known as upc, up, uc, u and CPU throughout the kernel's C code. They point to the connected process's major structures, plus contain the CPU number. They are defined in `sys/systm.h`.

T.TRACEM: T065: bit map controlling types of messages to place in kernel trace buffer.

B.KFTP: B060 pointer to kernel flow trace buffer (debugging feature)

Any flags set? At this point there could be 4 bits set in the register S1 copy of the XP flag bits: the one representing the original hardware event that brought the CPU out of user mode, plus the 3 pseudo-interrupts. Because of the leading zero count on S1 the CPU will jump to the handler for the interrupt type that is left-most in the flags register.

Their relative positions in this register are represented on the flowchart by the order of the interrupt handlers, from left to right (shown both in the Y-MP and C90 orders). The interrupt handler does not clear its bit from the XP (it is already cleared).

trapusr[]: This table is indexed by interrupt type and provides:

1. The address of the interrupt handler to jump to.
2. An ascii value for the handler to enter into the trace.
3. The signal number to use if the interrupt handler signals the connected process.

stack pointers: At this point the process has a stack pointed to by p_stack which is normally an area allocated in the kernel stack pool. We are about to jump into CAL routines. Most of them call C functions, thus requiring a stack. The stack is effectively cleared of all frames except one (umain). The code generated by SCC at the entry to each function assumes that registers B02, B66 and B67 are valid pointers to the current stack frame, the next available space on the stack and the end of the stack, respectively. (See detailed diagrams of the stack later in this chapter.) In effect, this CAL main loop executes in the context of umain()'s stack frame. The C portions of the kernel push more frames onto the stack, and then pop them. When they return to the main loop and it then exits to user mode the stack will be empty (except for umain()). If the interrupt handler sleep()'s, however, the CPU does a context switch to a different stack.

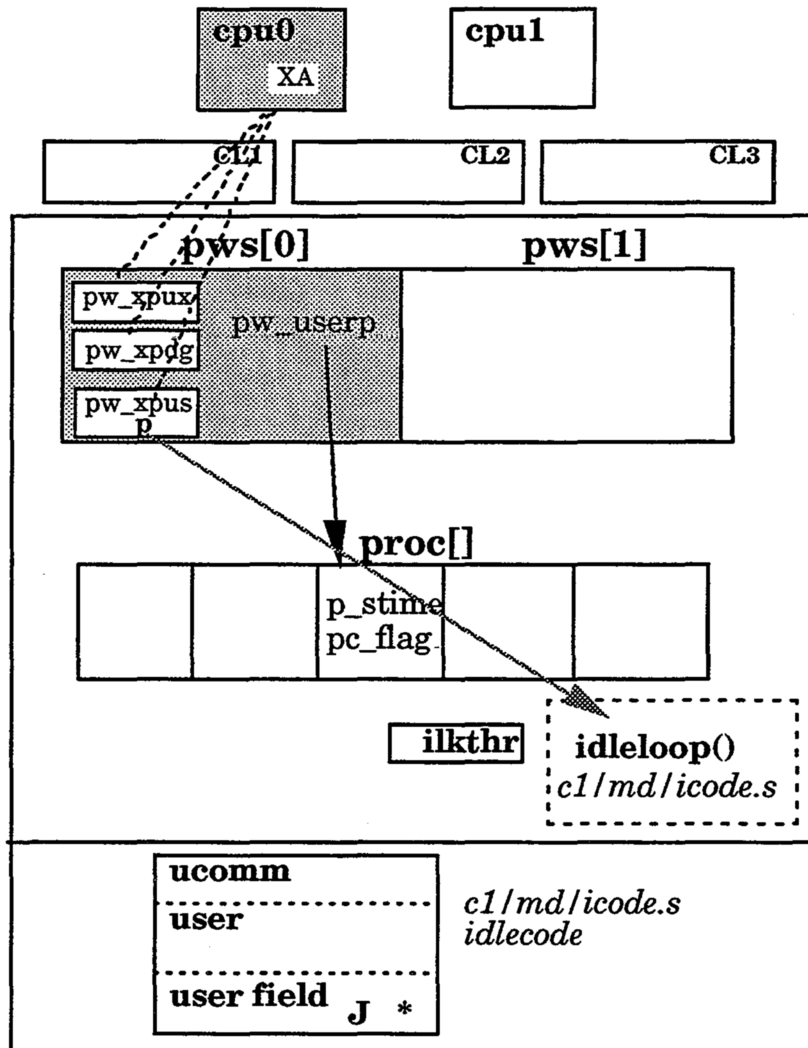
Idle processes

The diagram on the right shows those portions of the main loop relevant to the selection of a regular process versus the selection of an idle process and the function of the idle process. Processing relevant to "down CPU" is shown on following pages. Pictured below are the data structures involved.

Idle process selection: The test is (`pc_flag | PC_IDLE`). If the current process is the CPU's idle process, execute **`slaveidl`** else **`slaveusr`** goes through the testing for signals and restoration of user register context in preparation for exit to a user process (documented in following sections).

`slaveidl`: If the CPU is marked down or stopped (using `/dev/cpu ioctl`(2) call) `downcpu` is executed. Down CPU logic is shown on the following pages. The `pw_users` flag indicates a context switch is in order (switching away from idle in this case). `qswtch` logic is described with `slaveusr` logic.

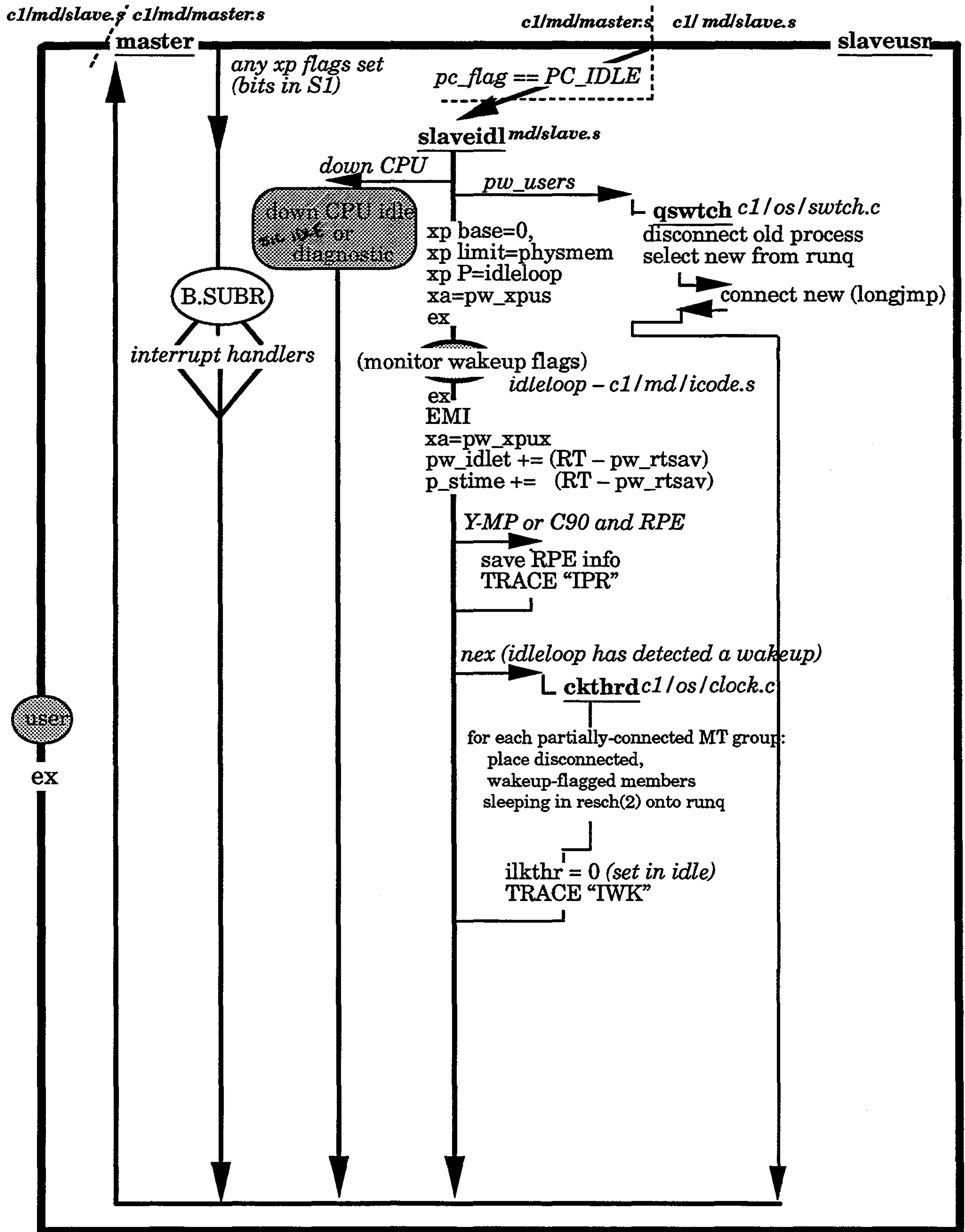
If the CPU is neither marked down or switching from idle, the "normal" CPU idle code is executed.



`idleloop`: An exchange package is prepared in the CPU's `pw_xpus` area to exchange to the `idleloop` code. The executable part of this idle process is the `idleloop` function in the kernel. All of user memory is addressable by this process. The CPU exchanges to this process without going through all the context save/restore of normal user processes. The CPU remains in this code until the following occurs: 1) an interrupt occurs, 2) a multithread monitor wakeup flag, tested every 010000 clocks, is detected. (multithreaded processing and `ckthrd` are described in "Process Management – Cooperative Parallel Interface").

Idle return to kernel: Idle connect and system time is computed for the idle process. Recall that each CPU has its own dedicated idle, timing is CPU specific. If the idle interrupted with `NEX` (monitor flag test) call `ckthrd` else loop to `master` to process interrupt.

Bored Idle Loop, Waiting for Something to do



Idle process - down CPU

Down CPU selection: A CPU can be set "down" or "stopped" with an `ioctl`(2) system call to `/dev/cpu` (see *UNICOS File and Formats and Special Files Reference Manual*, publication SR-2014 8.0 for more information.). The CPU is marked `PW_DOWN` or `PW_STOPPED` in its `pws` entry. This flag forces `swtch` to select the idle process to be connected to the CPU. `slaveidle` calls `downcpu` when either flag is set.

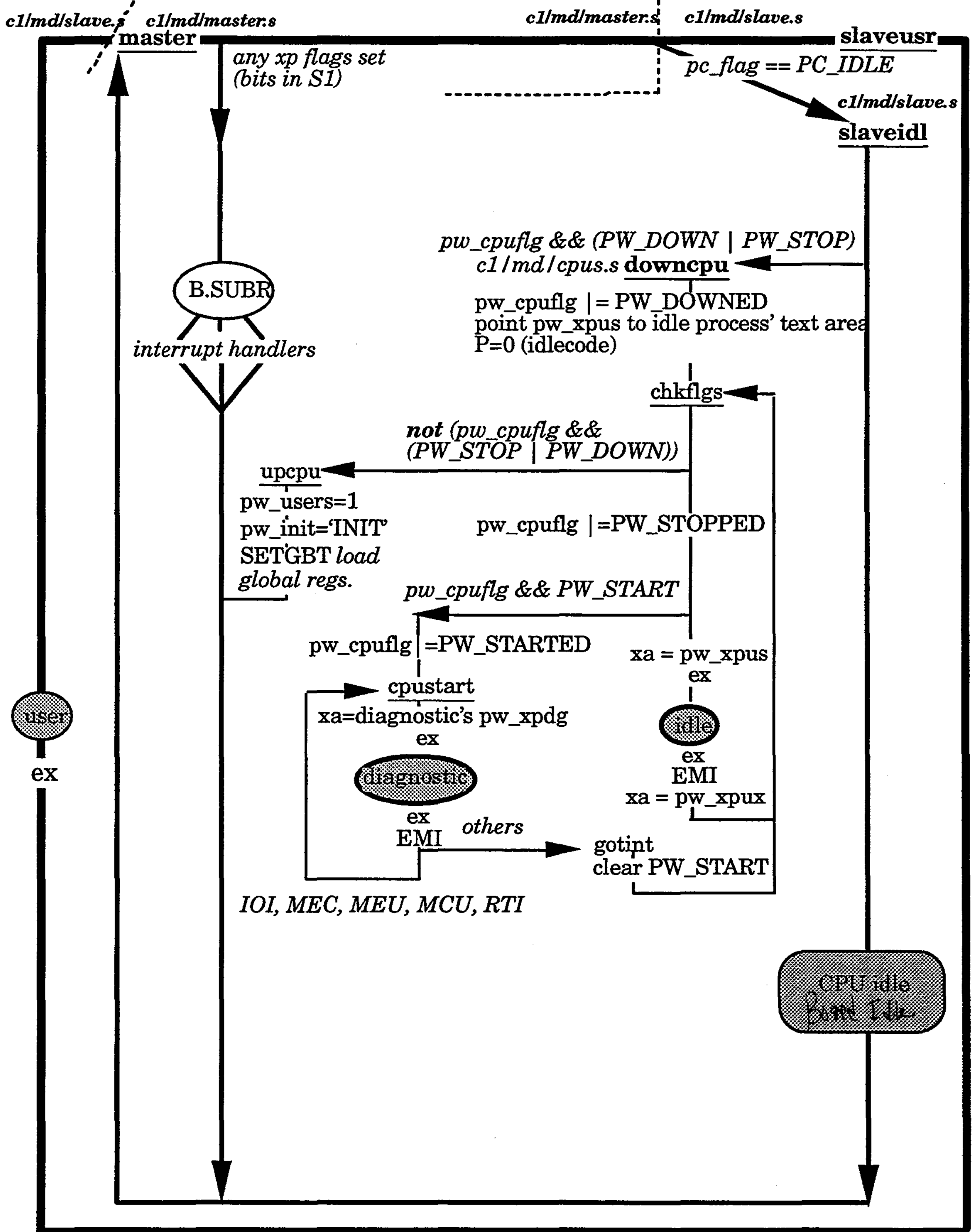
Down versus stopped: A CPU marked down is only allowed to execute the down CPU idle or a diagnostic (privileged) process. A CPU marked down and stopped executes the idle `idlecode` (pure idle, not thread test, and so on.). A down CPU can be given a diagnostic process (program) to execute (using `/dev/cpu` control) and started.

Diagnostic processing: The CPU exchanges and executes the diagnostic as if it were a "normal" user process but under strict control of the down CPU processing. CPU modes can be set by `/dev/cpu` driver. When the diagnostic completes usually an IPI issued by the diagnostic monitor process) the CPU returns to a **down** and **stopped** state. Additional diagnostics can be started.

Up CPU logic: A down CPU can be "up"ed by operator action through the `/dev/cpu` driver. The `PC_DOWN` and `PC_STOPPED` flags are cleared and the CPU executes `upcpu`. The `pw_users` flag is set to force the CPU through `qswtch` in order to select a process other than idle if one is qualified.

Note: Both the "idle CPU" `idleloop` and down CPU `idlecode` routines execute on behalf of the system processes proc[2], proc[3], and so on. Connect and system time is computed for the `idleloop` execution only. ♦

Idle Process - Down CPU



giveup() and idler

The kernel provides a mechanism to for a CPU to connect to its idle process. This is done in the following situations:

- **down_cpu()** c1/io/cpu.c : ioctl(2) to /dev/cpu, the CPU is forced to select slaveidl and proceed to idlecode.
- **profil()** c1/os/sys4.c: force the process through swtch() and resume() to reset the system "tick" rate to profile rate (1000 per sec default).
- **resch()** os/thread.c: force a CPU through process selection in resch(), and possibly idle if no other processes qualify for the CPU.

Function **giveup()** c1/os/slp.c provides this capability.

giveup logic: The diagram on the right illustrates giveup and idler logic. Table idler[] has one entry per configured CPU. Setting a non-zero value in idler[cpu] forces swtch() to select only the idle process for that CPU.

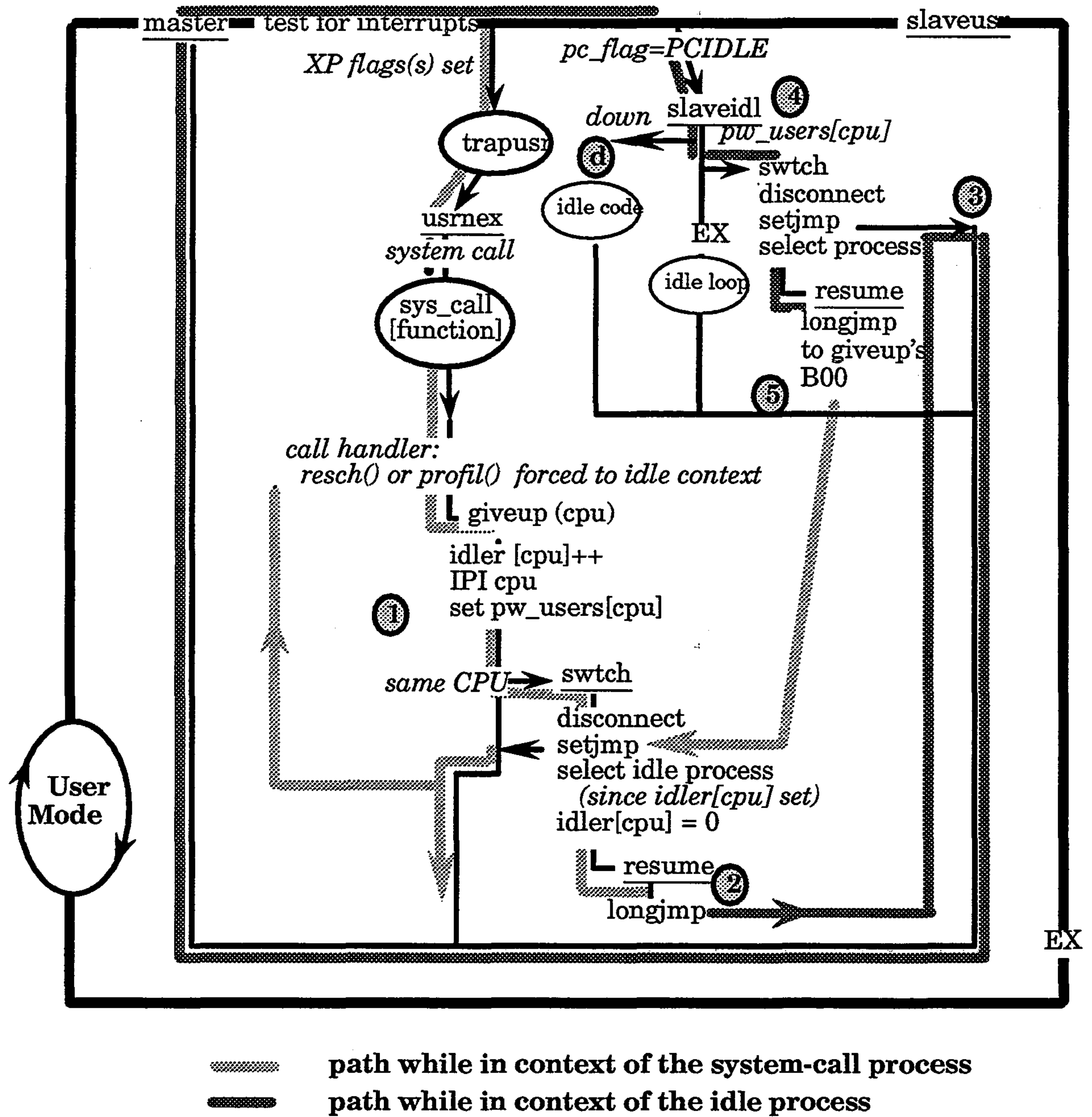
The diagram shows a CPU's path from a system call like profil(2) or resch(2) that calls giveup(). Logic proceeds from the system call through the master loop, and back again to the system call.

1. giveup() is called to force a trip through swtch().
   - The CPU's idler entry is flagged.
   - The CPU's pw_users is set to force a call to swtch() (at #4).
   - The CPU is interrupted with an IPI to force it from "User Mode" to "System Mode" if necessary.
   - The CPU calling giveup() is the one forced to idle call swtch().
2. Executing swtch() causes a longjmp() to the idle context (3).
   - The current process is disconnected, its context saved for later.
   - The idle process for this CPU is selected to run next, forced by the non-zero value in idler[cpu].
   - The flag in idler[cpu] is cleared.
   - resume() longjmps to idle (the saved context for an earlier call to setjmp() in slaveidl). We don't want to execute the idle loop (necessarily), we just want to make a trip through resume() to perform some scheduling action.
3. The CPU proceeds to master. After any interrupts are processed (under the context of this idle) the CPU proceeds to slaveidl.
4. Because pw_users is set, the CPU will call swtch() which:
   - Saves the context of the idle with setjmp().
   - Selects the "system calling" process for reconnection (it wins over idle).
   - Calls resume() to resume our process. **This call to resume is the reason giveup() was call in the beginning**.
5. The longjmp sends the CPU back to the system call logic, just after its call to giveup().

In the case of the ioctl(2) system call to down a CPU two actions may take place:

6. If the CPU executing ioctl() is the same one being set down, the logic proceeds as above except at (4) slaveidl "traps" the CPU and sends it to idlecode (d). (See description of slaveidl.)
7. If the CPU executing ioctl() is different than the one being set down, the IPI in giveup() will interrupt the CPU forcing it into the kernel (if it is not already there). The CPU executing ioctl() will NOT call swtch() but simply continue on. The "downed" CPU will be sent to slaveidl and then proceed to idlecode (d).

giveup() and idler



path while in context of the system-call process
path while in context of the idle process

Rung Evaluation

Process selection

At the right is the "upper right corner" of the kernel main loop diagram starting at the tag slaveusr cl/md/slave.s. This logic is executed unconditionally by any CPU headed back toward user mode with the following exceptions:

- In the SSD ssread/sswrite "hot path"
- The multitasking call 212 "very hot path"
- Connected to an idle process – CPU executed slaveidl.

There are 2 major considerations to check for before restoring the user hardware context and exchanging back to user mode:

1. Is the CPU connected to the right process?
2. Is there a signal to act on for this process?

The diagram on the right deals with the first question. Signal processing is dealt with in the next section.

Reasons for a context switch:

pw_users: Should be read as "p w user switch", also known as offset "W@SWAPF" in assembler code, and "SWAPF" in C code; and is frequently set with the SCHEDCPU macro in C code. This flag word is set by any function wanting some CPU to check the runq for a possible context switch. For example, a CPU might move a sleeping process back to the runq, check the pws table to find the CPU executing at the worst priority (worse than the newly connected process), and flag that CPU's pw_users and send an interprocessor interrupt to that CPU. (No hardware interrupt need be sent if the CPU has targeted itself.) Once the targeted CPU comes into the kernel it will always look at this flag, disconnect from its poor priority process and scan the runq for the better one. Other examples include the CPU scheduler lowering the priority of the connected process below another on the runq, or a connected process suspending itself.

PC_SSPND: The current process has suspended itself with a suspend system call or has been suspended by the memory scheduler. (Suspended processes stay on the runq, but are not connected.)

PC_FREEZE: The current process is frozen for a checkpoint or a debugger (using /proc). A process is frozen by calling freeze().

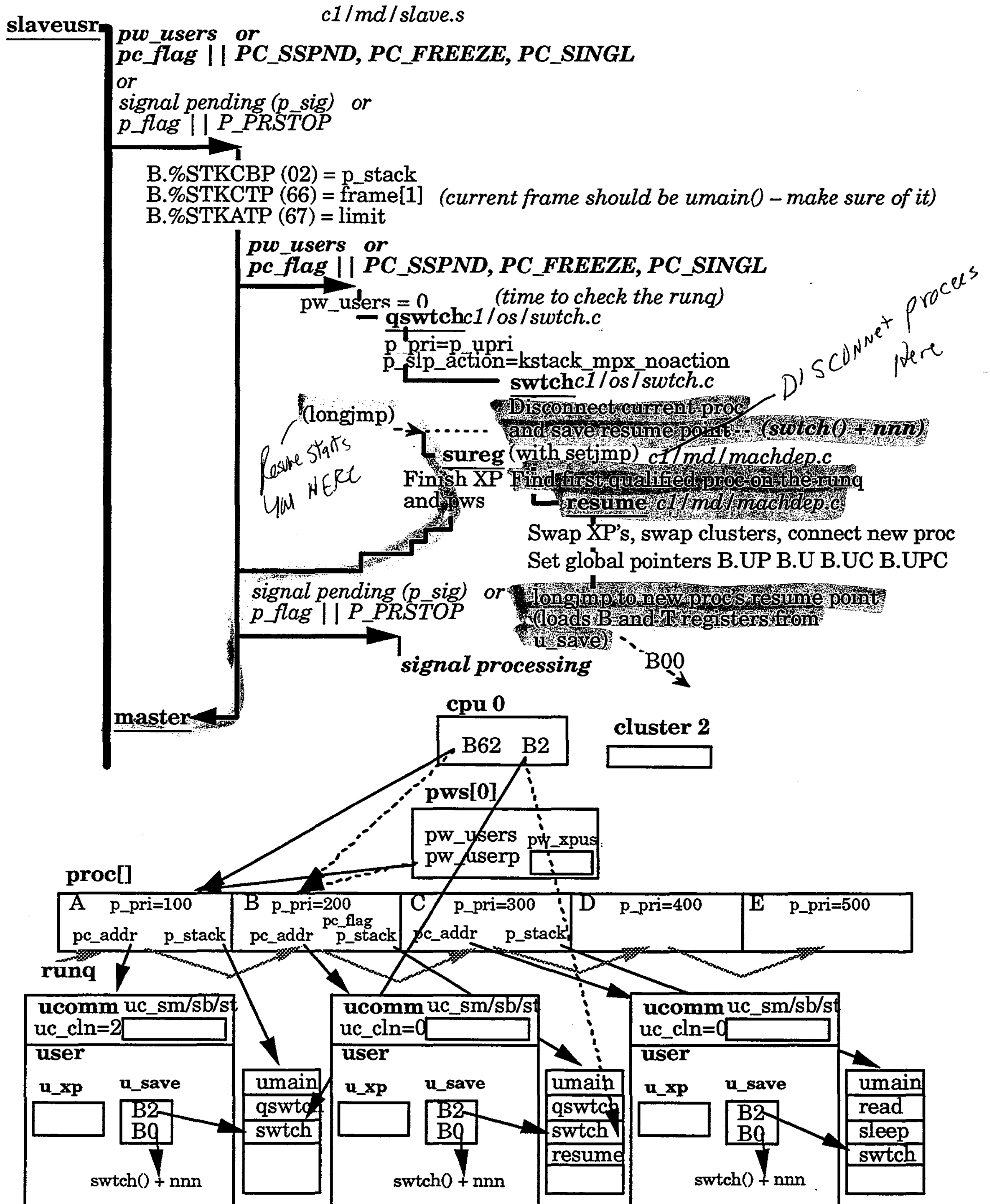
PC_SINGL: The current multitasking group is to be disconnected except for a single process. This is done by calling single() for operations like moving or expanding a multitasking group. For example, the memory scheduler calls single() to disconnect an entire group for a swap.

Context switch logic:

1. The stack is initialized at the base umain frame.
2. pw_users is cleared and qswtch() is called to disconnect the current process, select a new process to run from the sorted runq, and resumes the newly selected process.

   The newly selected process resumes execution – call old process is runnable on the runq waiting for a CPU but not connected. Note the stack shows qswtch() calling swtch().
3. The original process priority improves over time and eventually gets selected to be connected again by another process calling swtch().
4. The resumed original process checks for pending signals. If there are none the CPU loops back to master to check for any possible pseudo interrupts, and eventually returns to slaveusr. The process would normally NOT proceed through qswtch() immediately again would but continue through kernel exit and exchange to the user process. ***Note:*** *It IS possible that the newly resumed process loses its CPU after reconnection , before it gets a chance to exchange to the user, but this should be rare.*

Process Selection



Signals

Signal detection

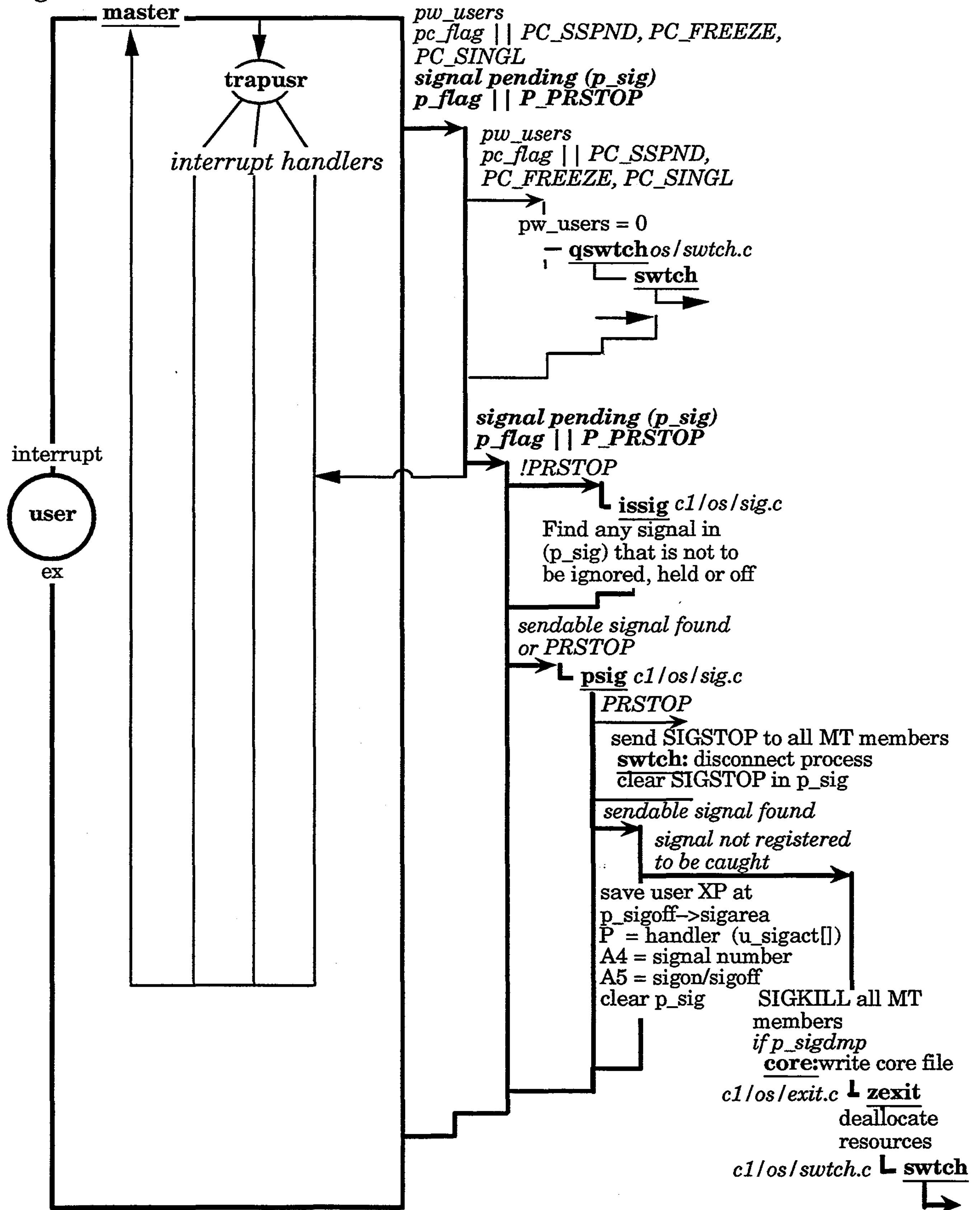
Here, at the point just before the exchange back to user mode, is where the kernel detects and acts on a signal. The implication of this is that a signal has an effect only upon a connected process. (As noted later in the Process Management section on `sleep()`, however, posting a signal in a process may make it connectable.)

A process that is not runnable or made runnable by sending it a signal cannot be killed by that signal.

- **`p_sig`** – Each bit is a signal. See `/usr/src/uts/include/sys/signal.h`.
- **`P_PRSTOP`** – This proc flag is set by an `ioctl(2)` to `/proc` requesting a process to be stopped immediately.
  - Set by an `exece(2)` when the process is to be stopped on exec (also a `/proc` `ioctl(2)`).
  - Set by `psig()` when a process is to be stopped on receipt of specified signals (also a `/proc` `ioctl(2)`).
  - Set when a multitasking group is to be brought down to a single process during an `exec(2)` (calls `mtcollapse()`).
  - The `P_PRSTOP` flag forces `psig()` to be entered to do any necessary processing (listed below).
- **issig()** – This functions tests for the presence of a processable signal and returns the number of the most "important" signal present and processable. `issig()` is explained in detail on the next page.
- **psig()** – This function performs the following:
  - Performs the `zexit()` call for multitasking members which are to leave on `exece(2)`.
  - Stops multitasking siblings on a traced signal.
  - Turns off any ignored signals which are not registered or held.
  - Determines the most processable signal remaining ("n").
  - If signal "n" is registered, the following events happen:
    - ◇ Calls `sendsig()` to alter the user XP as shown in the diagram.

      Else
    - ◇ If stopping (`p_sigstop`) signals are present stops the process (for example, remove from `runq` and `swtch()`) .
    - ◇ Writes a core file if dump-causing (`p_sigdmp`) signals are present.
    - ◇ `SIGKILL`'s multitasking siblings.
    - ◇ Calls `zexit()`.
- **zexit()** – Function `zexit()` is the "funnel" through which all processes leave the system, whether normally via the `exit(2)` call, or abnormally via a fatal signal. This function performs the following:
  - Deallocates all the kernel resources associated with the exiting process (except its "zombie" proc table entry, in which it preserves the process id and exit status).
  - Does a context switch off to some other process' stack. See the Process Management chapter on the subject of Process Termination for details.

Signal Detection



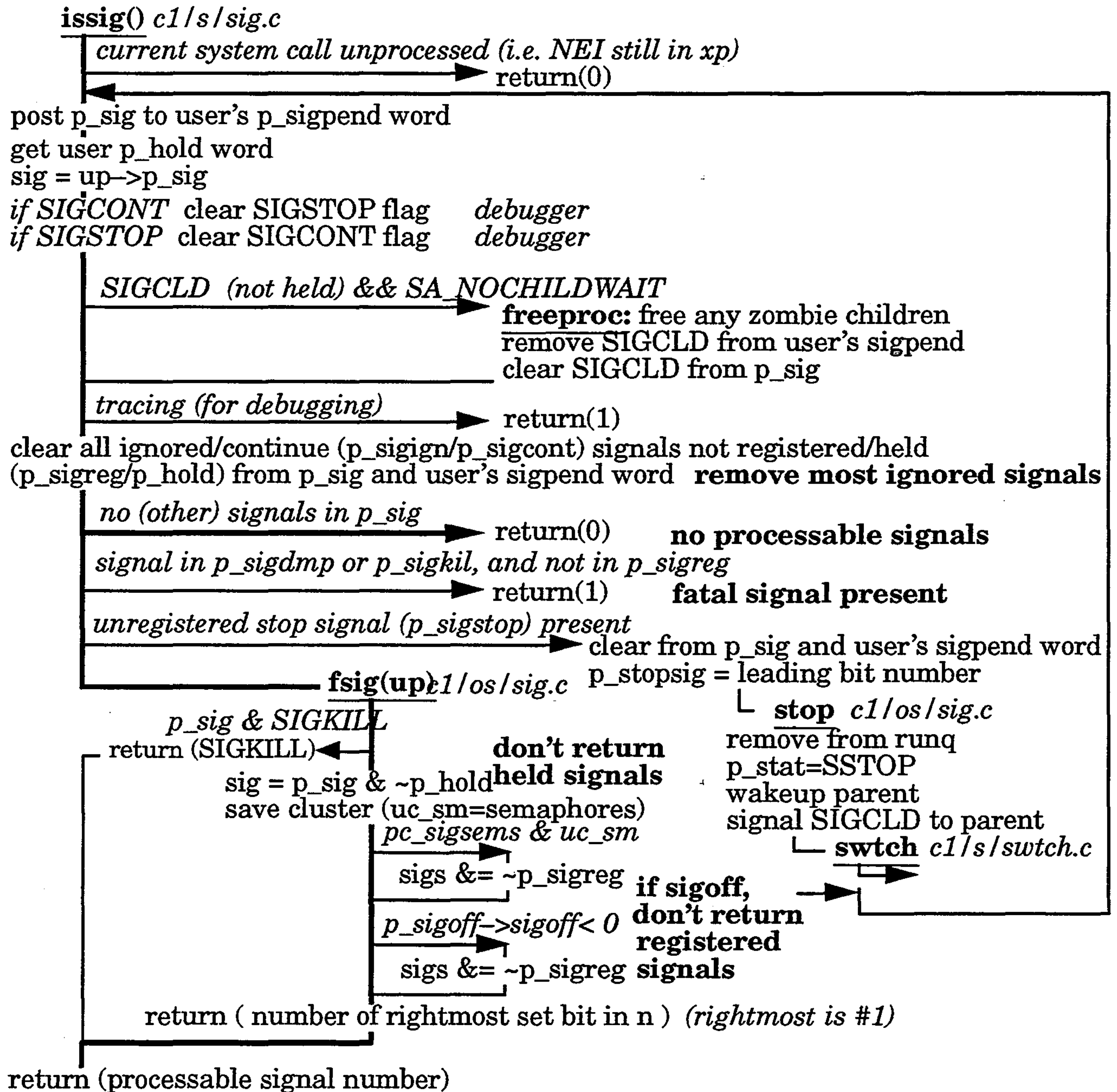
issig() - Kernel's test for a processable signal

The function issig() returns the number of the "most processable" signal present in p_sig. It returns zero if there are no processable signals.

This function is called by c1/md/slave.s, as shown on the previous page. It is also called by sleep() to test whether a sleep was interrupted by a signal or should return because a signal is already present.

- **NEI check**: As described earlier in this chapter, it is possible for a CPU to "see" several pending interrupts at once. The NEI flag may be present while executing some other interrupt type first. The interrupt flags in the xp are not cleared until the interrupt handler is about to be entered. So it is conceivable that a CPU might enter the kernel on a system call but call sleep() before usrnex is entered. No signal is to be returned to sleep() until usrnex has been entered. sleep() might do a longjmp using a bad context. See the details of sleep() in the "Process Management" section.
- **sigpend**: The user sigarea structure contains the "sigpend" word which the kernel is supposed to maintain as a copy of p_sig.
- **sigheld**: The user sigarea structure contains the "sigpheld" word which the kernel is supposed to maintain as a copy of p_sigheld.
- **SIGCLD**: This is the death-of-child signal. If a process is ignoring SIGCLD or registers it with a null catching function, zombies are created but exist only until the parent makes a trip through the kernel and calls issig().
- **Tracing / stopped / continue**: These are all related to debugging done through the /proc interface.
- **Ignored**: Ignored signal are removed from p_sig unless registered or held.
- **Hold**: Held signals are not removed from p_sig.
- **Fatal signals**: p_sigkil is a mask of fatal signals. p_sigdmp is a mask of fatal signals which also cause a core file to be written. See /usr/src/uts/include/sys/signal.h for #define's (SIG_KILDFL, SIG_DMPDFL, others) which define which signals default to which category.
- **fsig():** Finds the number of the most processable signal in p_sig.
- **SIGKILL**: The KILL signal cannot be ignored or held. It is always returned first. Otherwise the leftmost signal in p_sig is returned (bit positions are counted from the right, from 1).
- **sigoff mode**: Registered signals are not returned by C if the user is in sigoff mode. There are 2 ways to set sigoff mode:
  1. Set the sigoff word in the user sigarea to –1
  2. Register any semaphore as a sigoff indicator with $SIGNAL (SCTL_SEMA), then set that semaphore

issig() - Kernel's test for a processable signal



Catching a signal

At the right is an example of a C program catching a signal. The signal in the example is a SIGHUP, and the function which is to process the signal is "catch()".

This example provides a functional overview of the following:

- Interaction between user code and the signal libraries
  - Interaction between the signal libraries and the kernel
  - Signal library data structures
  - Kernel signal data structures

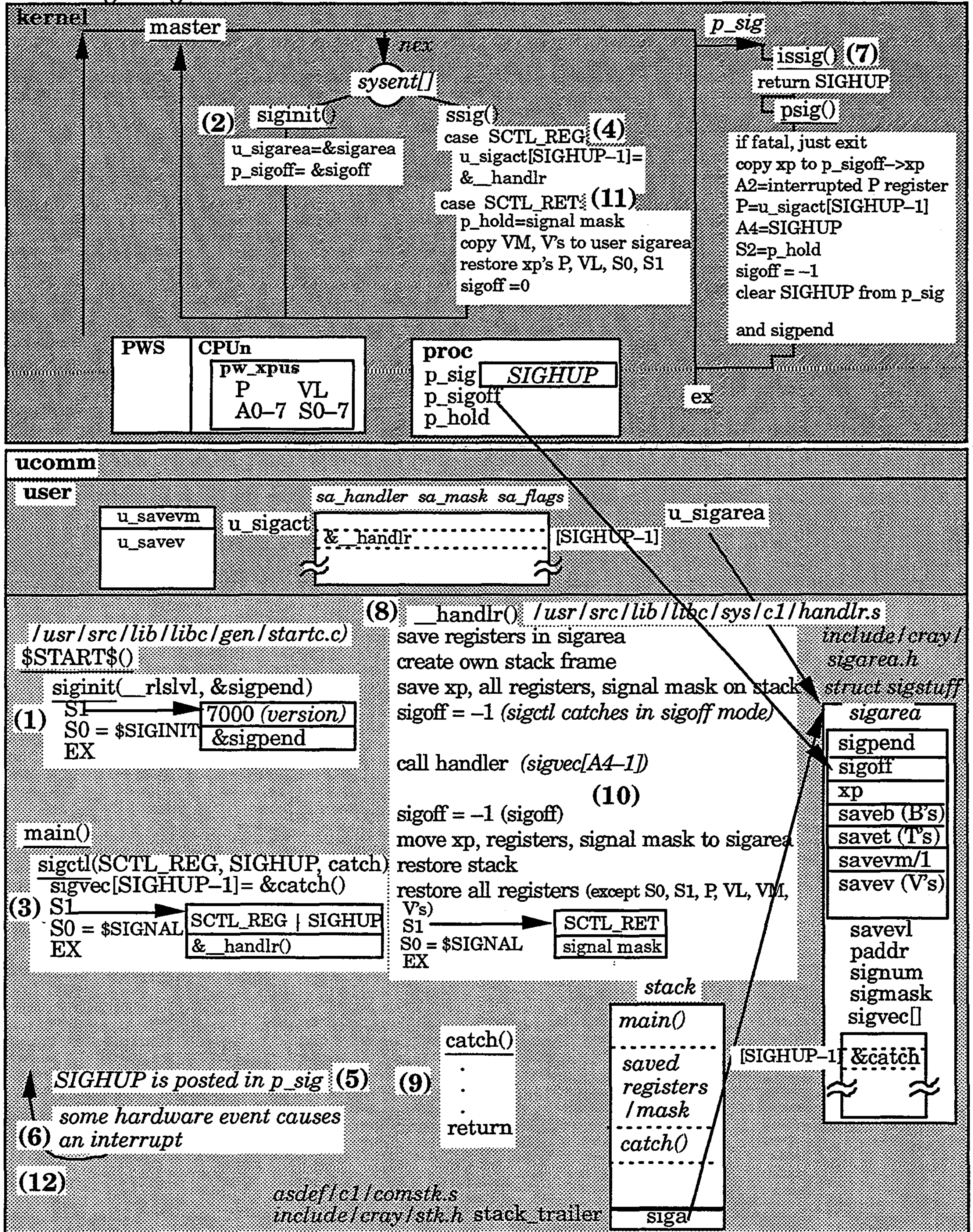
1. The startup module ($START$) allocates the "sigstuff" structure (with the sigarea structure within it) and records it address in the stack_trailer structure at the end of the stack. It calls the library function _siginit() to register the signal processing area (sigarea structure) with the kernel.
2. The kernel siginit() function saves the address of the area, then returns to user mode.
3. The application program (main() in this case) "registers" to catch the SIGHUP signal by calling the library sigctl() function. Other methods exist to either catch it or prevent it from killing this program. But sigctl() is one straight-forward way to handle the signal.

Note: The sigctl() function records the catching function's address "locally", in the sigvec table. Its $SIGNAL system call passes to the kernel the signal number and the address of the library signal catching routine __handlr(). ♦

4. The $SIGNAL system call is processed by the kernel function ssig(). The cpu then returns to user mode in main().
5. In the example, some other process posts a SIGHUP in this process' p_sig. But the posting of the signal does not interrupt the signaled process.
6. The signal will not be processed until a cpu enters the kernel while in the context of this process. Any interrupt to this cpu will cause it to exchange out of user mode and into the kernel. The application has no control over this.

   (However, the application does have control over whether or not the kernel will enter __handlr() *if a registered signal (the* SIGHUP *in this example) is present. This is the purpose of the* sigoff *word. If* sigoff *is negative, the process is in "*sigoff*" mode and the kernel will not process registered signals.)*

Catching a Signal



7. On the process' next trip through the kernel the presence of the SIGHUP bit in **p_sig** will be detected by the call to **issig()**. The issig() function returns SIGHUP (for example, "true") if the user is in sigon mode. The issig() function returns the rightmost signal in p_sig that is not being ignored, held nor is a registered signal deferred by the user's sigoff word. It also clears the ignored signals from p_sig unless they are registered or held.

   Because issig() returned "true" the **psig()** function again selects the rightmost signal in p_sig that is not being ignored, held nor is a registered signal deferred by the user's sigoff word. If the signal is fatal, for example, not registered, the process is terminated here. In the example, the SIGHUP is registered in the **u_sigact** table, so the kernel will alter the user exchange package (**p register**)to enter user mode in the signal catching function __handlr().

   But the key to reentering main() at the interrupted point must first be preserved. The **interrupted exchange package** is saved in the "xp" field of the user's sigarea structure. Recall that the address of this area was registered by the siginit() call.

   The user exchange package (in the pws[]) is modified so that the CPU will enter __handlr() and the SIGHUP will be available to __handlr() in register A4. The kernel also sets the process into sigoff mode so that another external interrupt will not cause __handlr() to be reentered while it is saving the present hardware context (of main()) and modifying its stack.

8. The CPU next returns to user mode in **__handlr()**. This function saves the balance of the interrupted hardware context in the "sigarea" (the kernel has saved only the xp). Because there is only one sigarea per process it must be saved on the stack in order to allow a signal catching function to be interrupted by a hardware event which then results in entry to another signal catcher. This "nesting" of signal processing is possible to any depth, limited only by the maximum size of the user's stack.

   The __handlr() function must do the "ticklish" work of allocating space on the user stack even though the present frame may be in a "half-pushed" or "half-popped state". It may have to allocate more stack now and deallocate it later. This work must be done in a **sigoff** mode.

   The __handlr() function uses its local sigvec table, indexed by the signal number passed to it by the kernel, to enter the catch() function.

   In this example, catch() is to be entered in **sigoff** mode because it was registered by the sigctl() function. If the catching function is to be entered in **sigon** mode (for example, interruptible by a signal), the sign bit of sigvec[signal number-1] would be set. The signal() and sigaction() functions would set the bit and __handlr() would set sigon mode before calling the catching function.

9. The **catch()** function is a regular C function. There is nothing special about it. It pushes its frame onto the stack at entry and pops it off when it returns.

10. The catch() function returns to __handlr(), which goes back to sigoff mode while it restores the stack and most of the interrupted hardware environment. sigon mode must be restored (the process had to be in sigon mode originally to be catching a signal) but this cannot be done while in __handlr() because this restoration process must be entirely complete. So __handlr() makes a $SIGNAL system call to "return" to the interrupted point.

11. The kernel ssig() function (SCTL_RET request) restores the balance of the interrupted hardware environment and sets the process back to sigon mode.

12. The exit from the kernel is back to the interrupted point in main().

Signal data structures

p_sig A signal is a bit in this word. "Signaling" a process is done by the kernel function psignal(). The bits are numbered from 1, and from right to left. Their names are defined in /usr/src/uts/include/sys/signal.h.

p_sigoff Records the (user relative) address of the sigon/sigoff mode word. The word is usually the sigoff word in the user's sigarea structure, and its address is "registered" by the siginit(2) call. **Caution**: the kernel sendsig() function assumes that the exchange package save area immediately follows the user's sigon/sigoff word.

p_hold Contains the "signal hold mask". The presence of a signal in this mask means that processing of this signal is deferred. The signal bit remains in p_sig even though it is not processed.

p_stopsig Contains the integer value of the signal that "stopped" the process. This occurs when a process is to be stopped for debugging on any of a set of signals kept in p_prsigmask and pc_prsigmask.

p_sigpend Records the (user relative) address of the user's "signals pending" word. The word is usually the "sigpend" word in a sigarea structure, and its address is "registered" by the siginit(2) call. The kernel keeps the indicated user word in sync with p_sig.

p_sigign Ignored signals. The presence of a signal matching a bit in this mask means that issig() should (normally) clear it from p_sig (unless it is registered or held).

p_sigkil Fatal signals. The presence of a signal matching a bit in this mask means that the process should (normally) exit (unless it is registered or held).

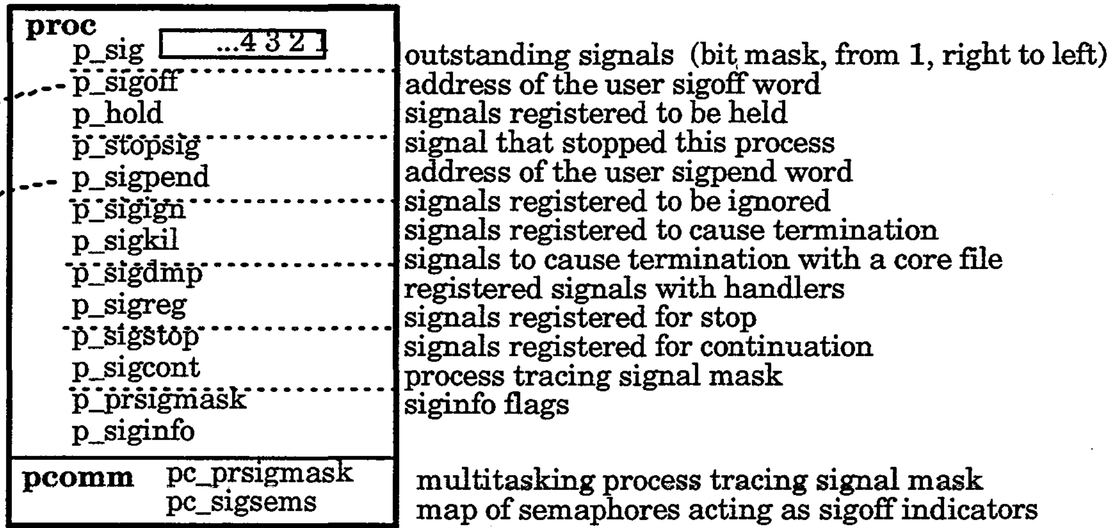
p_sigdmp Fatal signals producing a core file. The presence of a signal matching a bit in this mask means that the process should (normally) write a core file and exit (unless the signal is registered or held).

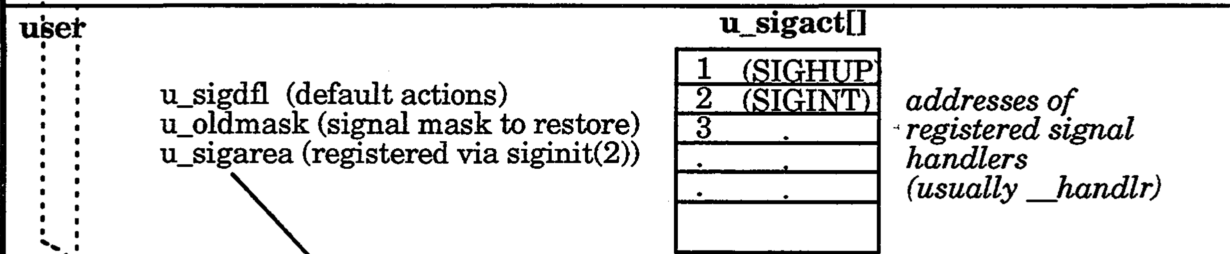
p_sigreg Registered signals. Any signal matching a bit in this mask has an action registered in u_sigact[] (either a handler or a flag (SA_xxxxx)).

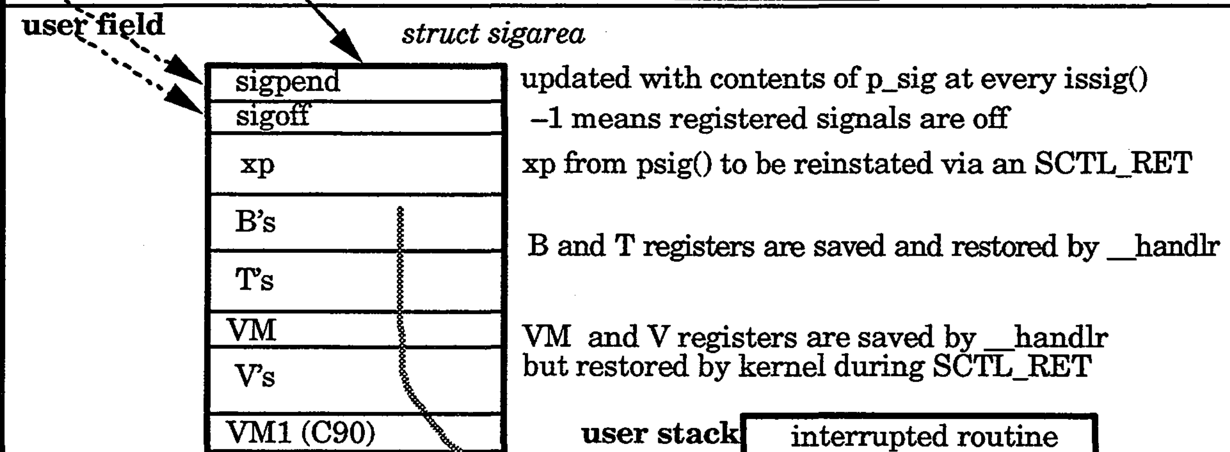
p_sigstop Debugging stop signals. Any (unregistered) signal matching a bit in this mask should cause the process to enter a debugging "stop" state.

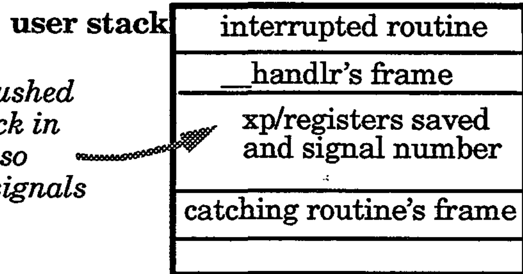
p_sigcont Debugging continue signals. Any signal matching a bit in this mask should cause the process to awaken from the debugging "stop" state

Signal Data Structures









Signal data structures

p_prsigmask Debugging trace signals. Any (unregistered) signal matching a bit in this mask should cause the process to enter a debugging "stop" state (set up by a /proc debugger).

p_siginfo A bit mask of signals sent via calls to qsignal() (quota.c). It is retrievable via the getinfo(2) system call.

pc_prsigmask Debugging trace signals. Any (unregistered) signal matching a bit in this mask should cause the multitasking process group to enter a debugging "stop" state (set up by a /proc debugger).

pc_sigsems Hardware semaphores used as sigoff indicators. If nonzero, fsig() unloads the hardware semaphores. Any semaphore matching a bit in this mask means that the process is in sigoff mode. This is a complement to the sigoff word (see p_sigoff above), but designed for multitasking efficiency.

uc_sm Semaphore register save area. The fsig() function unloads the hardware semaphores to this field (see pc_sigsems above).

u_sigdfl Default action signals. Any signal matching a bit in this mask is to be handled in the default manner. Used by sigaction() to set p_sigign, p_sigdmp, p_sigkil, p_sigstop, p_sigcont. Reset by setregs() on exec.

u_oldmask Saved copy of p_hold to preserve it during a sigsuspend(). Restored by psig() before entering a signal catcher.

u_sigarea This word records the (user relative) address of the user's sigarea. This area begins with the "sigpend" word, and its address is "registered" with a siginit(2) call.

u_sigact[] This array of "sigaction" structures (signal.h) contains the action to take for each registered signal (see p_sigreg). Each structure contains the following:

sa_handler optional user entry point (usually __handlr()).

If the sign bit is set, the handler is to be executed in sigon mode.

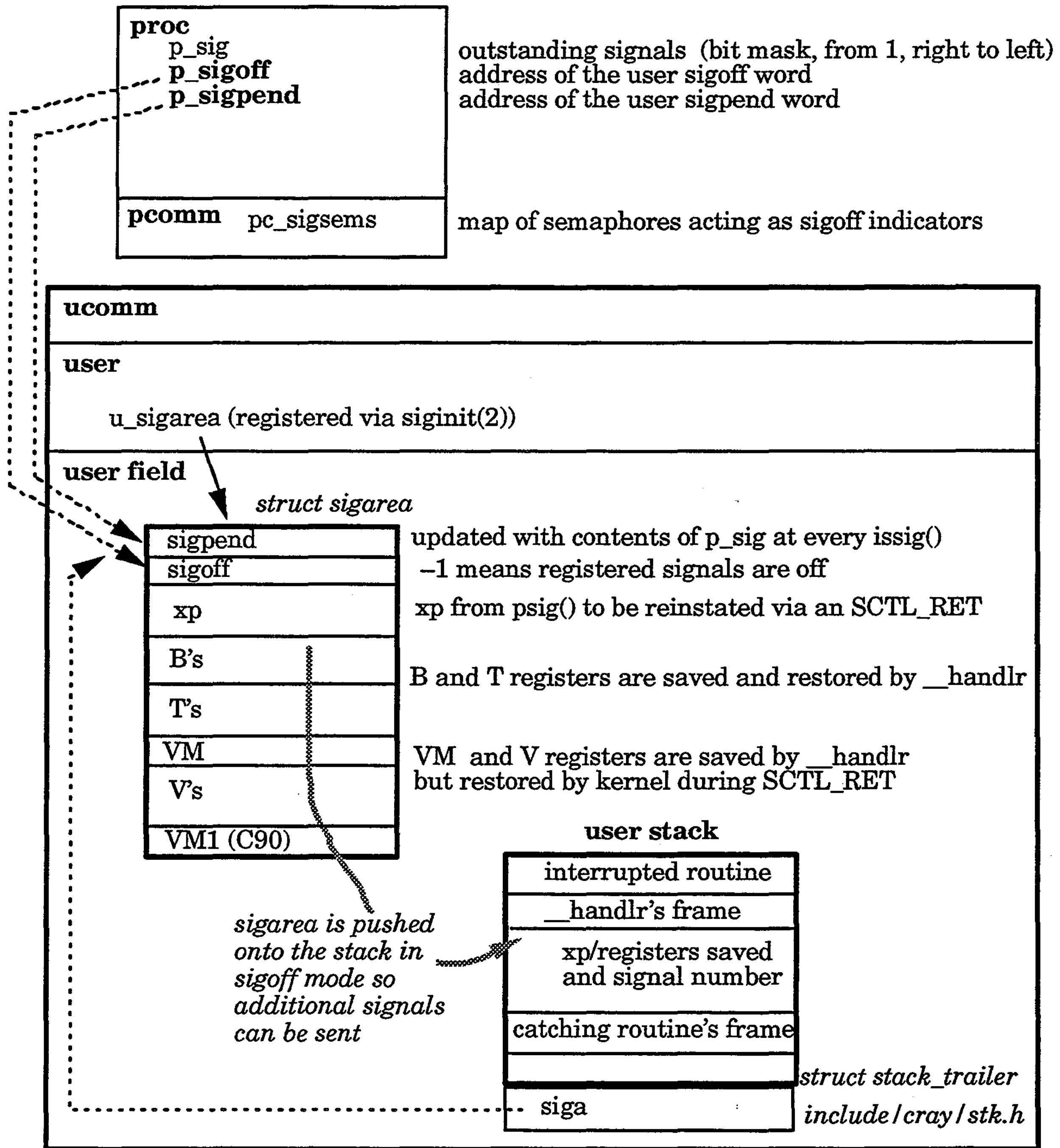
sa_mask becomes p_hold during execution of the catcher.

The previous p_hold is passed to the catcher in register S2.

sa_flags SA_xxxxx; such as SA_WAKEUP, SA_CLEARPEND.

These can be set by a user to request special handling by the kernel (see sigaction() in TR–USC).

Library routines words



These library routine words are found in the user-addressable `sigarea` structure:

`sigpend` Contents are equal to `p_sig`'s. (See `p_sigpend` above).

`sigoff` Indicates `sigon` mode (0) or `sigoff` mode (–1). (See `p_sigoff` above).

This page used for alignment

Kernel signal processing overview

The diagram on the right shows an overview of the kernel's main loop summarizing the system call handlers relating to signal processing and showing where signals are processed.

The "box" on the left side of the diagram is a summary of which library routines make the system calls.

The kernel call handlers (below) are coded in `c1/os/sys4.c`. CAL names for system calls are coded in `/usr/src/lib/asdef/c1/comsys.s`. See also `/usr/src/uts/include/signal.h`.

ssig() Handles the `$SIGNAL` call, with all of its SCTL_xxx varieties of requests.

sigprocmask() Handles the `$SIGPROCMASK` call.

sigpending() Handles the `$SIGPENDING` call.

sigsuspend() Handles the `$SIGSUSPEND` call.

sigaction() Handles the `$SIGACTION` call.

siginit() Handles the `$SIGINIT` call.

Sources of signals:

usrfpi() This is an example of a kernel function which posts a signal; in this case the floating point error interrupt handler. The process signals itself by calling **psignal()**. Other interrupt handlers generate signals, too.

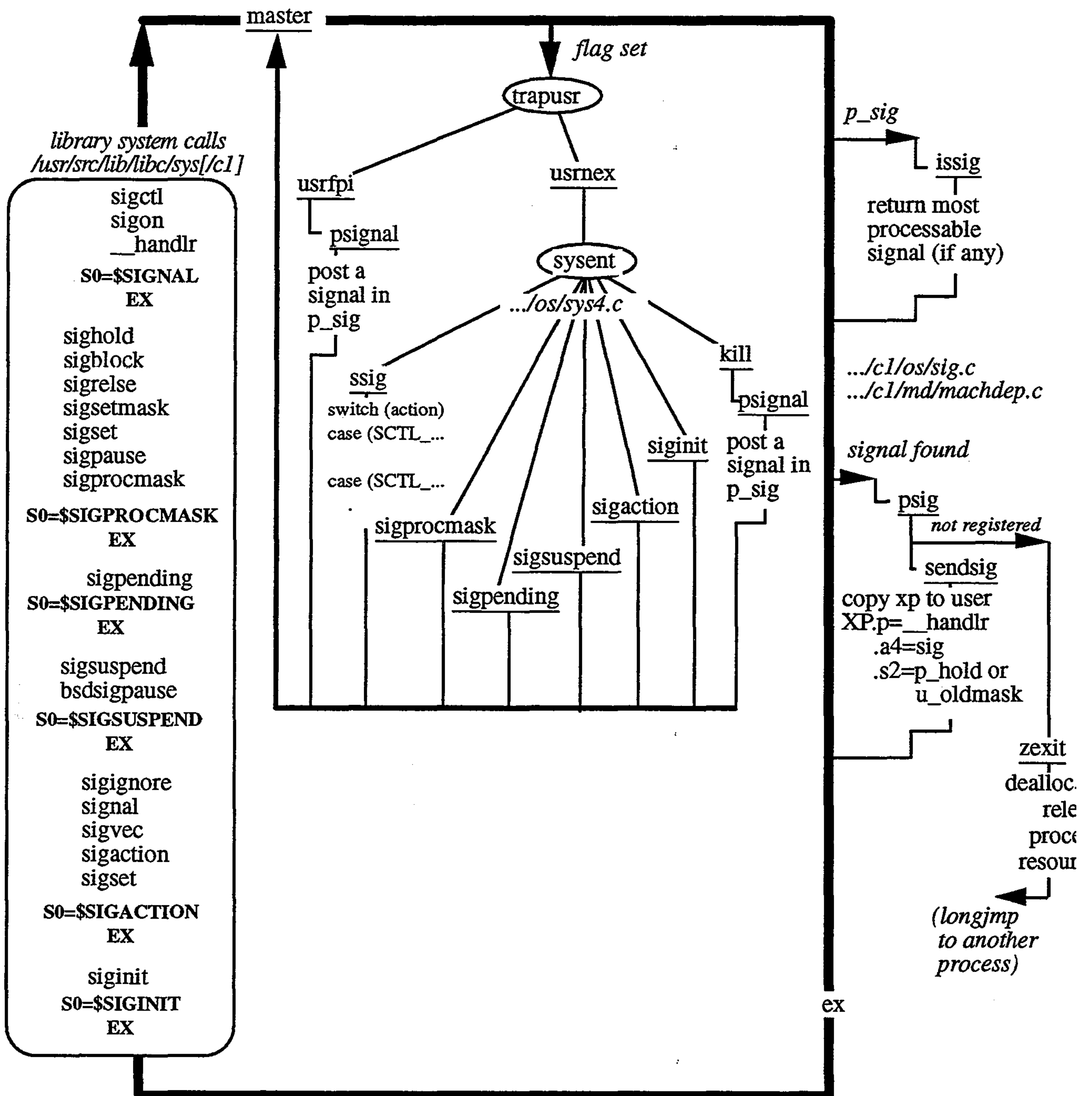
kill() This is an example of a kernel routine which posts a signal; in this case a system call. The process signals another process by calling **psignal()**.

Acting on a signal:

issig() (Detailed earlier) : Any process attempting to exit the kernel will check its `p_sig` field for the existence of a signal. If any are present, `issig()` is called to test if any are that can be processed. `issig()` generally returns the number of the lowest numbered signal that can be processed (but will return a "process tracing" signal, or `SIGKILL` first, if present).

psig() If `issig()` returned a nonzero value `psig()` will "process" the signal. Fatal signals not found to be registered will cause the process to exit. If a handler was registered (typically `__handlr()`) the exchange package is saved in user space, and the user's exchange package is modified to enter that handler and inform it of the signal number and any signal hold mask that should be restored after executing that handler.

Kernel Signal Processing Overview



Library signal processing overview

Summarized on the right are the library pathways to the 6 signal-related system calls. Source code for the signal library routines is in **/usr/src/lib/libc/sys[/c1]**.

The purpose of the diagram is to show which system call is used by each library routine to accomplish its purpose.

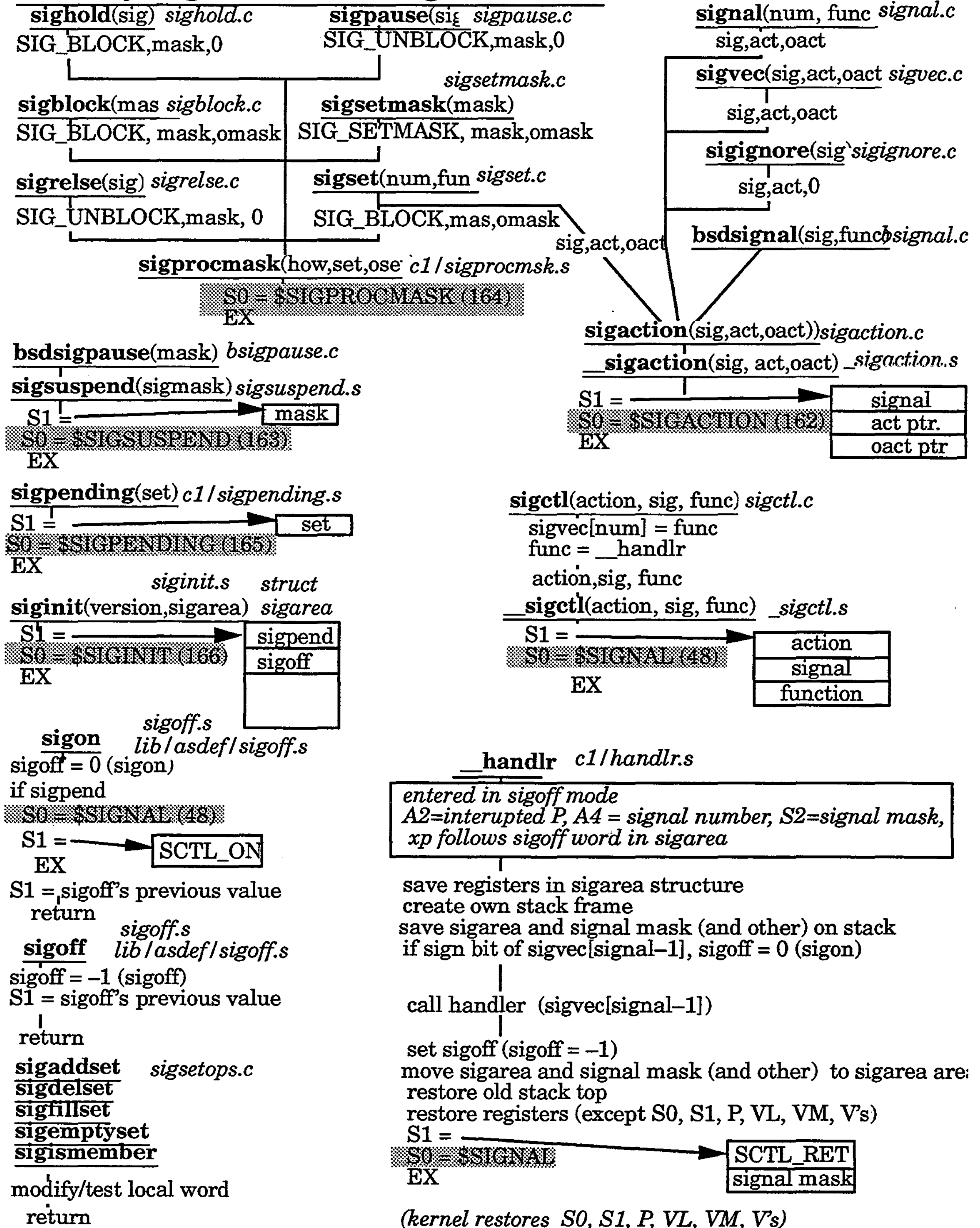
For a functional description of each routine refer to its man page and its description in the UNICOS System Calls course document (TR–USC).

Five of these system calls were new at release 6.0 (162–165 were implemented to conform to POSIX standards):

`$SIGACTION`	(162)
`$SIGSUSPEND`	(163)
`$SIGPROCMASK`	(164)
`$SIGPENDING`	(165)
`$SIGINIT`	(166)

Compatibity is maintained in the 7.0 kernel handler `ssig()` for the pre-6.0 signal libraries. The addresses of the "`sigoff`" and "`sigpend`" words may still be passed with the `$SIGNAL` call even though they are registered by the `$SIGINIT` call.

Library Signal Processing Overview



Kernel exit

Exit from the kernel is the "lower right corner" of the kernel main loop diagram. At this point the kernel need only **restore the user's hardware context** and do some **miscellaneous accounting**.

Pictured on the facing page are the main points in the logic of flow of that "exit housekeeping" and the data structures involved.

BMM: The Bit Matrix Multiply unit, if present in the hardware, works on a 64-word operand, loaded from a vector register. The kernel never uses the BMM unit, so we do not see the kernel save the BMM on entry to the kernel. But during a context switch or fork of a new process the functions `bmmsave()` or `bmmdmp()` save the BMM unit in `u_savebmm` and set the `u_bmmsaved` flag (see `c1/md/bmmsave.s`).

Vector registers: The Vector Length register is part of the exchange package, so it need not be considered separately.

The Vector registers are not preserved across a system call. (The compilers do not vectorize across a function call, so it is safe to assume that the user is not depending on the contents of the V's after calling a library system call function.) The NEX flag from the xp is preserved in `u_unex`. If the process is returning to user mode from a system call its V's must be zeroed because a context switch may have occurred during the call, resulting in some other process having loaded sensitive information into the V's and itself been disconnected. For the current process to have access to another process's V's would be a security hole. It is, however, much faster to zero out the V's than to save and restore them from memory.

If this was not a system call, the kernel may have saved 1, 2 or 8 V registers. Recall that at entry, if not a system call, the kernel saves VM in `u_savevm` (and, if a C90) the last 64 bits (VM1) in `u_savevm1`), V0 in `u_savev` and sets `u_vsaved` to 1. Any assembler function which needs to use 2 V's will save V1 and change `u_vsaved` to 2 (no C function uses any vector – the kernel is compiled in non-vectorizing mode.) If a process is disconnected the balance of the V's are saved during the context switch and `u_savev` is set to 8. The effects of those saves are reversed here.

Trace 'USR': The UNICOS memory-resident trace will show the tag 'USR' as the CPU exits to user mode. See the `/etc/crash` "`utrace`" directive.

Save of **u, uc, up:** These pointers to the current `user`, `ucomm` and `proc` structures (respectively) are actually B registers (see `systm.h`). They are stored in A registers to preserve them for the process's next entry to the kernel.

VM: The Virtual Machine feature allows UNICOS to run copies of UNICOS as processes. Hardware XP flags are saved in `pw_vmsav` for the virtual machine process.

Restore B's and T's: The kernel entry sequence unconditionally saved the user's B's and T's in `u_saveb` and `u_savet`.

Note: "1–63" here is decimal, for example, all of them are restored. (B00 is restored below). ♦

Kernel Exit

if BMM and u_bmmsaved
restore BMM unit from u_savebmm
NEX
restore V0 — V0 = 0
restore V1 or V1–7 — V1 or V1–7 = 0
restore VM/VM1
A1 = &pw_xpus
trace 'USR'
A2 = up
A3 = uc
A5 = u
if VM restore XP
flags
restore B1–63
restore T0–63

from SSD hot path → **CALLHOT**

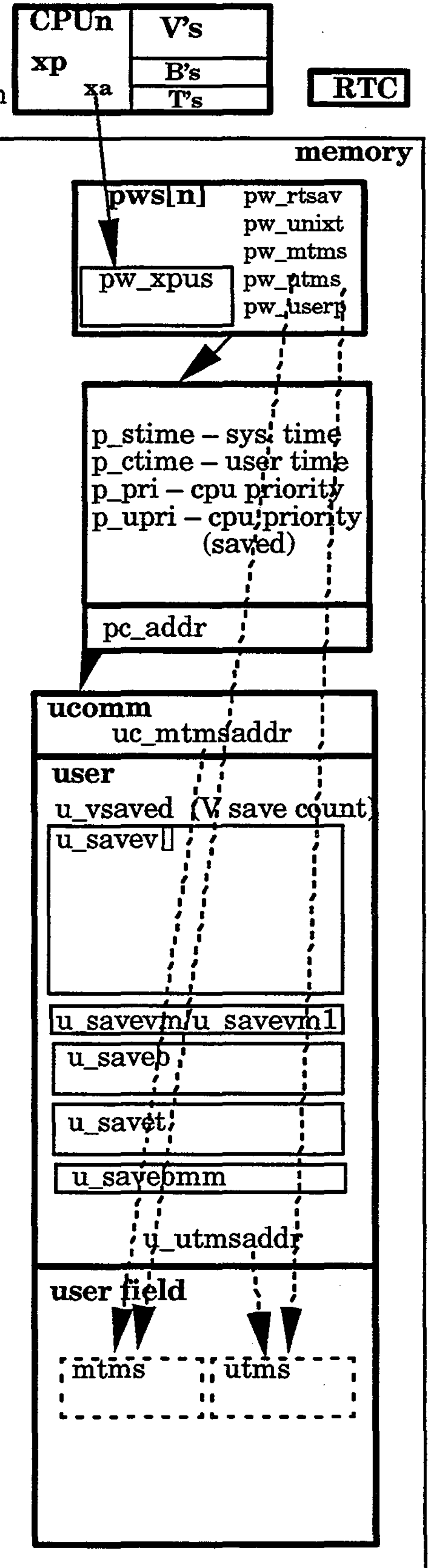
S4 = RTC
p_stime += (RTC – pw_rtsav)
pw_unixt += (RTC – pw_rtsav)
pw_rtsav = RTC
S5 = p_ctime.
(user) pw_utms–>utms_update = RTC
(user) pw_utms–>utms_utime = p_ctime
(from utimes() call)
p_pri = p_upri
pw_cpri = p_upri
update uc and pw_mtms–>mtms area
(from mtimes() call)

from atmxchg (S.C. 212) → **CALLHOTTER**

XA = A1 (&pw_xpus)
B00 = u_saveb[0]

EX
exchange to user mode

A1 = user xp address this cpu
A2 = up
A3 = ucomm addr
A4 = user-space mtms structure addr
A5 = u addr
A6 = pws entry address this cpu
A7 = unix xp address
S4 = RTC prior to exchange
S5 = p_ctime



Kernel exit (continued)

CALLHOT: The `ssread/sswrite` call "hot path" did not use and V, B, or T registers – it is all coded in assembler, and there is no possible context switch. So the exit from that logic can skip all restoration of the above.

system time: The real time from entry to the kernel (**`pw_rtsav`**) to this point is accumulated in the proc table (**`p_stime`**) and pws (**`pw_unixt`**) as system time. The real time at exit from the kernel is preserved in S2 for computation of user time at re-entry.

`pw_mtms` / `pw_utms`: These fields are relevant only if the connected process has made the `utimes`(2) or `mtimes`(2) sytem call, registering user-addressable areas where the kernel is to post timing statistics. (If the system call has not been made, the pointers point to "throw-away" areas.)

CPU scheduling priority: Field **`p_pri`** is the field used to order the `runq`. It is normally the process's scheduling priority calculated at some considerable expense by the fair share scheduler to a value between 60 and 999. But while a process sleeps it is assigned a "system" priority that affects how it will be awakened. Field **`p_upri`** simply preserves the user mode priority during a sleep, and that value is restored here.

The priority of each connected process is recorded in its CPU's pws entry (**`pw_cpri`**) to facilitate selecting a CPU to interrupt and send through a check of the `runq` when the `runq` is reordered or processes are added to it.

`CALLHOTTER`: The autotasking libraries make system call number 212 (the "auto-tasking mini exchange") to have the kernel restore a few registers for them (see the "kernel entry" section). This work is handled as a special case and is such a short path through the kernel that it is not even accounted as system time, hence this exit after all accounting procedures.

reloading the **XA:** Memory reads are complete at this point, so there should be no possibility of a memory error interrupt to the kernel. The eXchange Address register is pointed at the user exchange package (**`pw_xpus`**).

reloading **B00:** A hardware "return jump" instruction would alter the B00 register. But no more function calls will be done now. The user's B00 value is restored (`u_saveb[0]` has been prefetched into an A register).

exchange: The normal exit instruction ("ex") exchanges the contents of the CPU exchange package and `pw_xpus`. The kernel's A and S registers are thus preserved during the user interval in the "user exchange package".

Now that the CPU is in user mode, the CPU begins to increment the hardware performance counters again. These counters are only incremented in user mode. They are only read and reset in monitor mode.

In a C90, the exchange to user mode sets the Enable Interrupt Modes (EIM) flag, enabling all interrupts to this CPU which are flagged as enabled in the Mode register of this user XP.

This page used for alignment

Mainline inner loop - Interrupt handlers

usrnex - *User normal exit (System call)*

The NEX flag in the exchange package indicates a system call. The user calling sequence is:

S0	**= system call number**
S1	**= address of an argument list [optional]**
EX	

Typical return processing from a call:

if S0 != 0 (there is an error: u_error)	**Store S0 in errno**
(S1 and S2 are possible return values)	

For user system call examples, see /usr/src/lib/libc/sys/c1. For the names of kernel system call processors see /usr/src/uts/c1/os/sysent.c (processor code is in /usr/src/uts/os and /usr/src/uts/c1/os).

P_VM: If the calling process is a Virtual Machine process, copy the XP to that process; do not execute the call here.

Validation of user's call: The sysent table contains the number of arguments expected in a given call. usrnex() does not know what the arguments represent, but does validate that the user's S1 register does point to a valid argument list, for example, the whole list lies within the process' addressable field. The argument list itself is copied to the proc table p_arg array for use by the call processor. The user's first argument is assumed to be a pointer to a path name, and is saved in u_dirp (u_dirp is user-relative; the user's dba must be added to it before use). The field u_ap is loaded with the address of p_arg[] at the time of process creation.

Interruptible system calls: The u_save[QSAV] stack context is prepared to return to stjmp1 for an interrupted system call. See the "sleep" section in "Process Management" for detail on interruptible system call processing.

Mutitask single threading: In the multithreaded kernel all system calls are considered multi-threaded by default. Protection is provided by SEMLOCKs and MEMLOCKs as described in "System Initialization". However, only system calls flagged as MT in the sysent table are multithreaded among multitasked group members. Function mtasklock single threads on pc_mtask, setting p_mtask_locked in the lock "owning" process. The lock is cleared at usrnex01.

System call time: Current time before the start of each system call is stored in p_bsctime. If a context switch away from this process is done (by resume()) during the call, the elapsed time is added to p_sctime (system call time) and p_bsctime is zeroed. On reconnection, resume() saves the current time in pw_rtsav, which is used to update p_sctime instead of p_bsctime because p_bsctime is 0. Therefore system call time does not count sleep time.

Return values: Calls set RVAL1 and RVAL2 to reflect return values, including possible detail about error conditions. These are placed in the S1 and S2 areas in the user XP pw_xpus. The return error u_error is placed in user S0.

System call statistics: SCTRACE is a preprocessor option causing usrnex to update sysent[] fields sy_ncalls, sy_tottime, sy_maxpath, and sy_minpath. SCTRACE is defined as 1 in the released Nmakefile. Use "sar -c" to display system calls per second over a period of time. (The sar command reads the sysent table using /dev/kmem.) Other system all statistics are displayed with "sar -t" and "sar -H".

Cost of a system call: The number of system calls a user makes can be factored into his processes execution priority by the fair share scheduler. "kl_cost" is a field in the user's "lnode" and "shconsts.sc_call" is a dynamically tunable cost per system call (released as zero).

usrnex - User Normal Exit (System Call)

master — *test for interrupts* → *interrupt flag* → (trapusr[flag]) — PWS STUFF INTO HARDWARE

c1/md/master.s

usrnex

S3 = user S0 S4 = user S1
trace "NEX up/userS0 userS1"

p_pflag & P_VM → **vmtrap** *c1/os/vm.c* — copy the XP to the VM process → to usrnex02

user S0, RVAL1, RVAL2, and u_error = 0
validate function number (in sysent table bounds)
get sysent[function]

no arguments →

validate user arg list
copy arg list to p_arg[]
u_dirp = arg[0]
(u_ap == &p_arg[] already) — args live in Proc Table area

call is interruptable (36 of them are) — YES / NO

B00=stjmp1
quick stack context save
in u_save[U_QSAV]
(B00, B01, B02, B066, B067, T00)

(longjmp on sleep interrupted by a signal returnes here)

stjmp1:
if u_error = 0 → u_error=EINTR
p_intproc != 0 → ***p_intproc**(p_intarg) — interrupted system call processing
→ to usrnex01

stjmp2:

member of multitasked group AND call is not multi-threaded (22 of them are) → **mtasklock** *c1/md/machdep.c* — spin wait for pc_mtask — set

if PFROZE call **qswtch**

p_bsctime = RTC
p_syscall = call #
B00 = usrnex01

(sysent[syscallno].sy_call (p_arg,p_rval)) *c1/os/sysent.c* — getting an interrupt during a system call

205 system call handlers

sleep — *interruptable and processable signal* — jump B00

usrnex01

member of multitasked group AND call is not multi-threaded → **mtaskunlock** *c1/md/machdep.c* — clear pc_mtask — unlock

user S1 = RVAL1
user S2 = RVAL2
user S0 = u_error

if over 20 errors this process (u_errcnt) set pw_users

usrnex02

calculate statistics *(if SCTRACE)*
update p_sctime *(useing p_bsctime; sleep time is not counted)*
p_syscall = 0
shconsts.sc_csyscall ++
proc's kl_cost += shconsts.sc_syscall *(cost of a system call)*

p_pflag & P_VM → preserve XP flags

→ master

System entry table

Number : Description	Name	Args	INT/MT	Function()
0:indir; inop	illegal	0	0	nosys
1:exit	_exit	1	0	rexit
2:fork	fork	0	INT	fork
3:read	read	3	INT	read
4:write	write	3	INT	write
5:open	open	5	0	open
6:close	close	1	0	close
7:wait	wait	0	INT	wait
8:creat	creat	2	0	creat
9:link	link	2	0	link
10:unlink	unlink	1	0	unlink
11:exec	exec	2	0	exec
12:chdir	chdir	1	0	chdir
13:time	time	0	MT	gtime
14:mknod	mknod	11	0	mknod
15:chmod	chmod	2	0	chmod
16:chown	chown	3	0	chown
17:break	sbreak	2	0	sbreak
18:OLD stat pre 5.1	oldstat	2	0	oldstat
19:lseek	lseek	3	0	seek
20:getpid	getpid	0	MT	getpid
21:mount	mount	8	0	smount
22:umount	umount	1	0	sumount
23:setuid	setuid	1	0	setuid
24:getuid	getuid	0	MT	getuid
25:stime	stime	1	0	stime
26:ptrace	ptrace	4	0	ptrace
27:alarm	alarm	1	MT	alarm
28:OLD fstat pre 5.1	oldfstat	2	0	oldfstat
29:pause	pause	0	INT	pause
30:utime	utime	2	0	utime
31:for DFS	afs_syscall	6	0	afs_syscall

Number : Description	Name	Args	INT/MT	Function()
32:was gtty	x (gtty)	0	0	nosys
33:access	access	2	0	saccess
34:nice	nice	1	MT	nice
35:getinfo	getinfo	4	0	getinfo
36:sync	sync	0	0	syssync
37:kill	kill	2	0	kill
38:was switch	x (switch)	0	0	nosys
39:setpgrp	setpgrp	1	0	setpgrp
40:machine targeting	target	2	0	target
41:dup	dup	1	0	dup
42:pipe	pipe	0	0	pipe
43:times	times	1	MT	times
44:prof	profil	5	0	profil
45:proc lock	plock	1	0	lock
46:setgid	setgid	1	0	setgid
47:getgid	getgid	0	MT	getgid
48:sig	sigctl	4	0	ssig
49:IPC msgs; inop	x (ipc msg)	6	0	nosys
50:turn sacct off/on	jobacct	1	0	sessacct
51:turn acct off/on	acct	1	0	sysacct
52:IPC ShMem; inop	x (shmem)	4	0	nosys
53:IPC Sem; inop	x (sem)	5	0	nosys
54:ioctl	ioctl	3	INT	ioctl
55:oldlistio	x (listio)	0	0	nosys
56:user panic	upanic	1	0	upanic
57:uname	uname	1	MT	uname
58:reserved for USG	x (usg)	0	0	nosys
59:exece	exece	3	0	exece
60:umask	umask	1	0	umask
61:chroot	chroot	1	0	chroot
62:fcntl	fcntl	3	0	fcntl
63:ulimit	ulimit	2	MT	ulimit
64:ustat	ustat	2	0	ustat
65:lchown	lchown	3	0	lchown

Number : Description	Name	Args	INT/MT	Function()
66:was logins	x (logins)	0	0	nosys
67:was boot	x (boot)	0	0	nosys
68:set the time of day	settimeofday	2	0	settimeofday
69:get the time of day	gettimeofday	2	MT	gettimeofday
70:tfork	tfork	0	0	tfork
71:resch	resch	2	INT|MT	resch
72:Change Memory	chmem	2	0	chmem
73:listio	listio	3	INT	listio
74:thread	thread	1	0	thread
75:getpermits	getpermit	2	0	getpermit
76:setpermit	setpermit	3	0	setpermit
77:setfflg	setfflg	2	0	setfflg
78:setdevs	setdevs	2	0	setdevs
79:was setuint	x (79)	0	0	nosys
80:was getulvl	x (80)	0	0	nosys
81:was setulvl50	x (81)	2	0	nosys
82:getgroups	getgroups	2	0	getgroups
83:setgroups	setgroups	2	MT	setgroups
84:was setsysl	x (84)	0	0	nosys
85:setflvl	setflvl	2	0	setflvl
86:setfcmp	setfcmp	2	0	setfcmp
87:setfacl	setfacl	3	0	setfacl
88:was setucmp50	x (88)	2	0	nosys
89:was setusrv50	x (89)	5	0	nosys
90:getusrv	getusrv	1	MT	getusrv
91:slgentry	slgentry	2	0	slgentry
92:secstat	secstat	2	0	secstat
93:was getsysl	x (93)	0	0	nosys
94:was getfcmp	x (94)	0	0	nosys
95:getfacl	getfacl	3	0	getfacl
96:rmfacl	rmfacl	1	0	rmfacl
97:fsecstat	fsecstat	2	0	fsecstat
98:settfm	settfm	1	0	settfm
99:getsysv	getsysv	2	0	getsysv

Number : Description	Name	Args	INT/MT	Function()
100:tabinfo	tabinfo	2	0	tabinfo
101:tabread	tabread	4	0	tabread
102:suspend	suspend	2	INT	suspend
103:resume	resume	2	INT	ususpend
104:reada	reada	5	INT	reada
105:writea	writea	5	INT	writea
106:trunc	trunc	1	0	trunc
107:nicem	nicem	3	0	nicem
108:accounting ID	acctid	2	0	acctid
109:change SDS	ssbreak	1	INT	ssbreak
110:read from SDS	ssread	3	0	ssread
111:write to SDS	sswrite	3	0	sswrite
112:used to be usngl	x (112)	0	0	nosys
113:used to be uendsngl	x (113)	0	0	nosys
114:was idlep	x (idlep)	0	0	nosys
115:cpu/memory limits	limit	4	MT	limit
116:file pre-allocation	ialloc	5	0	iallocu
117:setsid	setsid	0	0	setsid
118:setpgid	setpgid	2	0	setpgid
119:cpu select	cpselect	2	0	cpselect
120:select	select	5	INT	select
121:category kill	killm	3	0	killm
122:recall async i/o	recalla	1	INT	recalla
123:getjtab	getjtab	1	MT	getjtab
124:setjob	setjob	2	0	setjob
125:mtimes	mtimes	1	MT	mtimes
126:checkpoint	chkpnt	4	0	chkpnt
127:recovery	restart	2	0	restart
128:utimes	utimes	1	MT	utimes
129:quotactl	quotactl	3	0	quotactl
130:set schedular vars	schedv	2	0	schedv
131:get system conf info	sysconf	1	0	sysconf
132:get path config info	pathconf	2	0	pathconf
133:get fd path info	fpathconf	2	0	fpathconf

Number : Description	Name	Args	INT/MT	Function()
134:OLD limits pre 7.0	limits_60	2	INT	limits_60
135:waitjob	waitjob	1	0	waitjob
136:rmdir	rmdir	1	0	rmdir
137:mkdir	mkdir	2	0	mkdir
138:getdents	getdents	3	0	getdents
139:statfs	statfs	4	0	statfs
140:fstatfs	fstatfs	4	0	fstatfs
141:sysfs	sysfs	3	0	sysfs
142:device accounting	devacct	3	0	devacct
143:dmmode	dmmode	1	MT	dmmode
144:used to be olddmofrq	x (144)	0	0	nosys
145:disk file account id	chacid	3	0	chacid
146:setusrv	setusrv	1	0	setusrv
147:stat 5.1	stat	2	0	stat
148:fstat 5.1	fstat	2	0	fstat
149:offline file req	dmofrq	5	0	dmofrq
150:setsysv	setsysv	2	0	setsysv
151:setfcls	setfcls	2	0	setfcls
152:setfcat	setfcat	2	0	setfcat
153:setucls	setucls	1	0	setucls
154:setucat	setucat	1	0	setucat
155:waitpid	waitpid	3	INT	waitpid
156:setucmp	setucmp	1	0	setucmp
157:setulvl	setulvl	1	0	setulvl
158:recalls	recalls	2	0	recalls
159:rename	rename	2	0	rename
160:enable/disable acct	dacct	2	0	dacct
161:write acct record	wracct	4	0	wracct
162:sigaction	sigaction	3	0	sigaction
163:sigsuspend	sigsuspend	1	INT	sigsuspend
164:sigprocmask	sigprocmask	3	0	sigprocmask
165:sigpending	sigpending	1	0	sigpending
166:siginit	siginit	2	0	siginit
167:accept	accept	3	INT	accept

Number : Description	Name	Args	INT/MT	Function()
168:bind	bind	3	INT	bind
169:connect	connect	3	INT	connect
170:gethostid	gethostid	0	0	gethostid
171:gethostname	gethostname	2	0	gethostname
172:getpeername	getpeername	3	INT	getpeername
173:getsockname	getsockname	3	INT	getsockname
174:getsockopt	getsockopt	5	INT	getsockopt
175:listen	listen	2	INT	listen
176:recv	recv	4	INT	recv
177:recvfrom	recvfrom	6	INT	recvfrom
178:was o2recvmsg	x (o2recvmsg)	0	0	nosys
179:send	send	4	INT	send
180:was osendmsg	x (osendmsg)	0	0	nosys
181:sendto	sendto	6	INT	sendto
182:sethostid	sethostid	1	0	sethostid
183:sethostname	sethostname	2	0	sethostname
184:setregid	setregid	2	0	setregid
185:setreuid	setreuid	2	0	setreuid
186:setsockopt	setsockopt	5	INT	setsockopt
187:shutdown	shutdown	2	INT	shutdown
188:socket	socket	3	INT	socket
189:socketpair	socketpair	4	INT	socketpair
190:symlink	symlink	2	0	symlink
191:readlink	readlink	3	0	readlink
192:lstat	lstat	2	0	lstat
193:sesscntl	sesscntl	3	0	sesscntl
194:reserved for CRI	reserved194	0	0	nosys
195:Res. for site use	x (195)	0	0	nosys
196:Res. for site use	x (196)	0	0	nosys
197:Res. for site use	x (197)	0	0	nosys
198:Res. for site use	x (198)	0	0	nosys
199:Res. for site use	x (199)	0	0	nosys
200:get device number	getdevn	2	0	getdevn
201:recvmsg	recvmsg	3	INT	recvmsg

Number : Description	Name	Args	INT/MT	Function()
202:sendmsg	sendmsg	3	INT	sendmsg
203:lsecstat	lsecstat	2	0	lsecstat
204:fsync	fsync	1	0	fsync
205:fchmod	fchmod	2	0	fchmod
206:fchown	fchown	3	0	fchown
207:vfork	vfork	0	0	vfork
208:exctl	exctl	1	MT	exctl
209:getlim	getlim	2	MT	getlim
210:setlim	setlim	2	MT	setlim
211:share sched control	limits	2	INT	limits
212:MXCHG – CAL syscall	MXCHG	1	0	nosys
213:getsectab	getsectab	2	0	getsectab
214:adjust the time	adjtime	2	0	adjtime
215:join files	join	2	0	join
216:fjoin files	fjoin	2	0	fjoin
217:set port bitmap	setportbm	1	0	ssetportbm
218:get port bitmap	getportbm	1	0	sgetportbm
219:tfork/thread/siginit	tfork2	2	0	tfork2
220:set file PAL	setpal	3	0	setpal
221:get file PAL	getpal	3	0	getpal
222:get proc privileges	getppriv	2	0	getppriv
223:compare priv text	cmptext	2	0	cmptext
224:set proc privileges	setppriv	2	0	setppriv
225:set file PAL	fsetpal	3	0	fsetpal
226:get file PAL	fgetpal	3	0	fgetpal
227:get mount info	getmount	2	0	sgetmount
228:pty reconnect	ptyrecon	2	0	ptyrecon
229:getpid	newgetpid	0	0	newgetpid
230:exit	newexit	1	0	newexit
231:kill	newkill	2	0	kill
232:category kill	newkillm	3	0	killm
233:site user exit	uesyscall	3	0	uesyscall

This page used for alignment

`usrioi` - *I/O interrupt*

Hardware principles of I/O interrupts:

IOI: CRAY Y-MP/X-MP: An I/O interrupt can occur at any time. If channel completion actually caused an exchange in a non-monitor mode CPU, the IOI bit is set in the exchange package. But if a channel completes a transfer while a CPU is already in monitor mode, the channel will direct the interrupt to this CPU (refer to the IOI bit's description in the Hardware chapter). The I/O interrupt remains pending for this CPU until it either returns to user mode or clears the pending interrupt via a machine instruction. The effect is that a single CPU in the kernel can handle all pending I/O interrupts.

IOI: CRAY Y-MP C90: An I/O interrupt can occur only when the SIE hardware "gate" or "flag" is set, which is when all CPUs are in user mode, or when a CPU in monitor mode executes the ESI instruction. In UNICOS this is done after handling all pending I/O interrupts, shortly before going back to `ioreenter`. The kernel executes with I/O interrupts disabled but will notice pending I/O interrupts by reading the CI register.

Interrupts are not directed toward a CPU in monitor mode, as they were in machines prior to the C90. CPUs in the UNICOS kernel do not have I/O interrupts enabled, so are not preferred for the interrupt. The lowest numbered CPU in user mode will get the interrupt. Such a CPU has both I/O interrupts enabled and the EIM flag set (it is set by the exchange to user mode). But the user mode CPU will not get the interrupt until any CPU in monitor mode executes the ESI instruction to set the SIE gate.

The effect is very similar to previous machines: a single CPU in the kernel can handle all pending I/O interrupts. The difference in a C90 is that a second channel completion (after the read of the CI yielded a zero, which resulted in the kernel doing an ESI) would cause an interrupt to a user mode CPU. On a previous machine the interrupt would have been directed to the monitor mode CPU, and remain pending until it exchanged to user mode.

UNICOS principles of handling I/O interrupts

General:

Only one CPU is needed to handle all pending I/O interrupts. The interrupt handler is single-threaded in the sense that only one CPU can execute it. Other CPUs noticing a pending I/O interrupt (by reading the CI) should ignore it. Generally, ST.IOFLAG (ST6) should keep all but 1 CPU out of `usrioi` processing.

The summary below describes general processing of each channel type. Additional detail on the drivers can be found in "I/O Management".

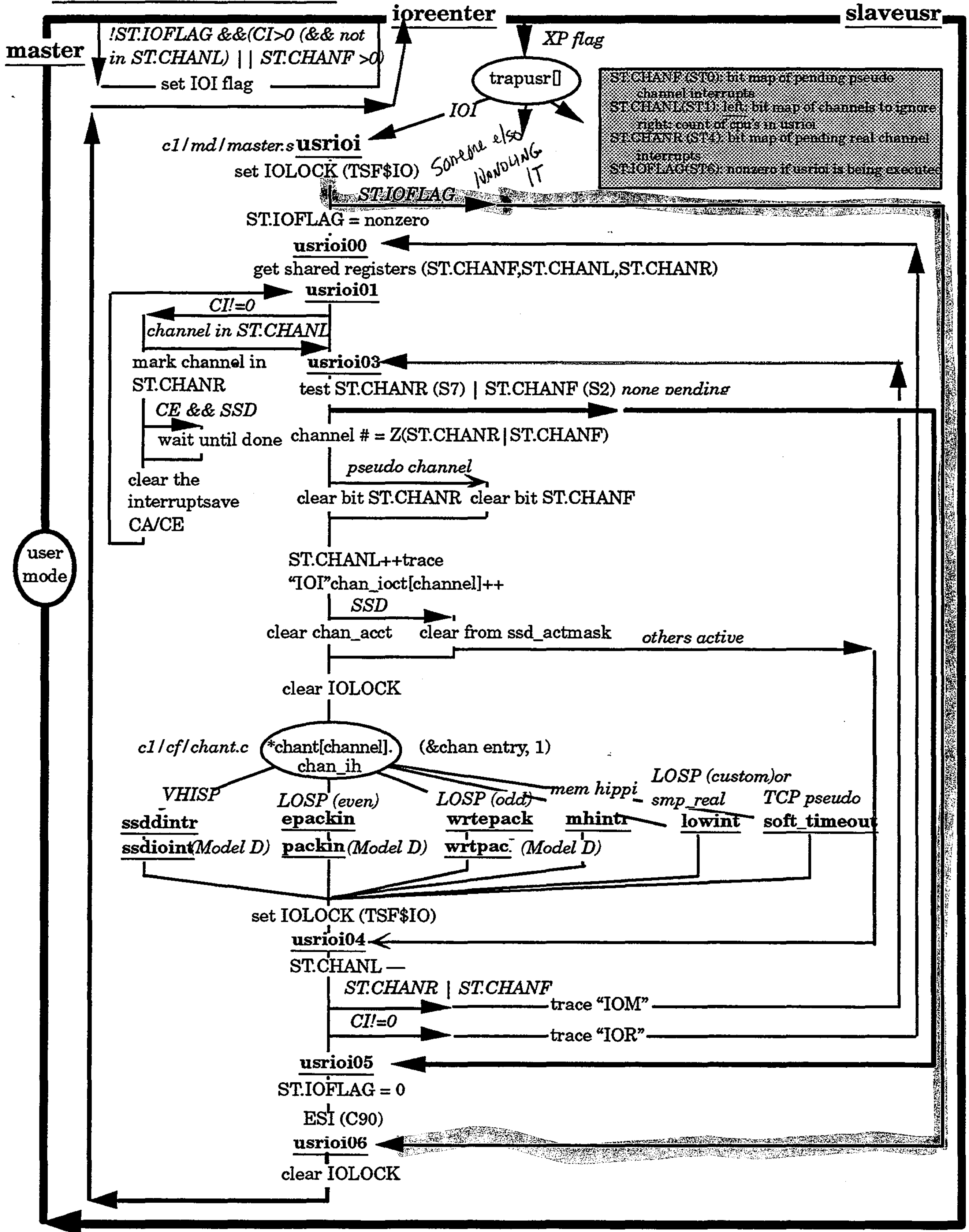
VHISP Channels:

VHISP channels are typically connected to SSD devices. These channels both read and write data under the control of the mainframe CPU. The UNICOS kernel SSD driver divides each I/O buffer into segments of equal sizes directing each configured and "up" VHISP channel to perform the read or write for each same buffer segment. The I/O is not considered complete until each channel completes and posts an interrupt. `usrioi` accumulates interrupts on all active channels until the last one completes before it considers the buffer transfer complete.

Channel lockout: The CPU may receive the interrupt from the channel before the actual I/O is completed. The CPU will spin waiting for completion of a transfer on the VHISP(s). While it waits, it sets the VHISP channel(s) into the ST.CHANL (ST1) bit map so that other CPU's will ignore VHISP interrupts.

When the last of the set of VHISP channels posts its interrupt `usrioi` calls `ssddintr()` (IOS E) or `ssdioint()` (Model B/C/D). These routines typically call wakeup() for the user waiting on the transfer and post any async I/O status and signals.

usrioi - I/O Interrupt



LOWSP Channels:

The typical use of lowspeed channels is packet I/O.

Interrupts on even numbered LOWSP channels represent packets sent by the IOS, usually indicating the completion of a UNICOS I/O request.

Interrupts on odd numbered channels reflect that a packet has just been sent to the IOS. The kernel checks if any other packets are queued to that channel and calls `wrtepack()` (IOS E) or `wrtpack()` (IOS models B, C and D) to initiate the writing of the next queued packet.

Memory Hippi: (Y-MP EL):

Y-MP El channels 024 026 040 042 044 046 060 062 064 066 0100 0102 0104 0106 support the memory device I/O. These channels are processed as real.

`smp_real`:

The `smp_real` is a LOWSP channel connected to the semaphore device supporting shared file systems (shared among multiple Cray Research systems). This is a special purpose channel used only for this function.

Pseudo Channels:

TCP/IP uses ST.CHANF (ST0) to indicate the completion of a transfer, either to delay processing until all real channels are handled (as TCP/IP).

Logic overview: simplified logic of `usrioi`, detail on following pages.

if another CPU already processing I/O interrupts leave to mainline.

while unprocessed I/O interrupts:

- poll CI merging all interrupts into a bit map word (real channel interrupts)
- merge real and pseudo interrupts
- for each bit in the bit map
  - select lowest numbered channel (in bit map) for processing
  - call corresponding handler to process the interrupt
  - clear the interrupt (channel and bit)