

KRYS SUPPLEE



IRIX Kernel Internals

Student Handbook

Part Number: TR-IKI-0.7-6.5-S-SD-W
SGI Proprietary
July 1998

RESTRICTION ON USE

This document is protected by copyright and contains information proprietary to Silicon Graphics, Inc. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of Silicon Graphics, Inc., is strictly prohibited. The receipt or possession of this document does not convey the rights to reproduce or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part, without the specific written consent of Silicon Graphics, Inc.

Copyright© 1998 Silicon Graphics, Inc. All rights reserved.

U.S. GOVERNMENT RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure of the data and information contained in this document by the Government is subject to restrictions as set forth in FAR 52.227-19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and /or in similar or successor clauses in the FAR, or the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor / manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., P.O. Box 7311, Mountain View, CA 94039-7311.

The contents of this publication are subject to change without notice.

PART NUMBER

TR-IKI-0.7-6.5-S-SD-W, July 1998

RECORD OF REVISION

Revision 0.7, Version 6.5, March 16,1998

SGI TRADEMARKS

IRIX, Silicon Graphics, and the SGI logo are registered trademarks of Silicon Graphics, Inc.

OTHER TRADEMARKS

Other brand or product names are the trademarks or registered trademarks of their respective holders

The contents of this publication are subject to change without notice.

IRIX 6.5 Kernel Internals (IKI65)

TR-IKI rev 0.7b SGI Proprietary (22jul1998)

Table of Contents

IKI: IRIX Kernel Internals Home Page

IKI65: IRIX 6.5 Kernel Internals	1
1 Training Materials	1
2 Training Material Utilities	2

IRIX Software Training

1 IRIX Software Training	1-1
1 Contents	1-1
2 Class Materials (SGI Employee Use Only)	1-2
1 I65RU	1-2
2 IKI65	1-2
3 OPET	1-2
4 PESTO	1-2
5 IFO	1-2
3 Reference Materials	1-3
1 Tech Digest links to many useful items ...	1-8
2 Internal Support Tools	1-11
1 Other Internal Support Tools	1-12
4 Cellular IRIX	1-14
5 Mail & Newsgroups	1-15
6 Performance	1-16
7 Performance Co-Pilot (PCP)	1-17
8 Application Programming	1-18
9 Hardware Reference Materials	1-20
10 Other Reference Materials	1-21

Cray Origin2000 Architecture

2 Cray Origin2000 Architecture	2-1
--------------------------------	-----

TR-IKI rev 0.7b SGI Proprietary

22jul1998

1 Cray Origin2000 Architecture Module ...	2-2
2 CRAY Origin2000 Multirack System	2-4
3 Router and Hypercube Connection	2-5
4 Hypercube	2-6
5 Origin2000 redundant paths	2-7
6 Module and Node Block Diagram	2-8
7 Node Board Components	2-9
8 Node Board, XBOW, and Router ...	2-11
9 MIPS @ R10000 Microprocessor (block ...)	2-12
10 More About the @ R10000 Microprocessor	2-13
11 More About Memory	2-14
1 Cache memory systems	2-14
2 Origin2000 Distributed Shared-Memory ...	2-15
3 Origin2000 Memory Hierarchy Diagram	2-16
4 Origin2000 Memory Hierarchy Explanation	2-17
12 More About Cache	2-18
1 Non-Blocking Cache	2-18
2 Cache Types	2-19
1 Primary Data Cache	2-20
2 Primary Instruction Cache	2-22
3 Secondary Cache (for Data and ...)	2-23
13 Determining What Hardware the System is ...	2-24
14 Determining What Memory Looks Like	2-25

Memory and Addressing: Pages, TLB's, ...

3 Memory and Addressing from a Hardware ...	3-1
1 HARDWARE MEMORY	3-2
2 Pages, and TLB's	3-2
1 Introductory Concepts About Pages	3-2
2 Introductory Concepts About the TLB	3-3
3 Memory Management Philosophies	3-5
1 Real Memory Machines and Swapping	3-5
2 Virtual Memory Machines and Paging	3-6
1 Where Are the Addresses the Process ...	3-7
4 Memory pages	3-8

TR-IKI rev 0.7b SGI Proprietary

22jul1998

i

5	HARDWARE ADDRESSING	3-9
6	All Addresses = (Page Number + Byte ...	3-9
7	Cray Origin2000 Memory Hierarchy and ...	3-10
8	Address Request Sequence	3-11
9	TLB Misses	3-13
1	Two Types of "TLB Miss"	3-14
10	TLB Size	3-15
11	Coprocessor 0 and the TLB	3-16
12	Binary, Hexadecimal, and Decimal ...	3-18
13	The 64-Bit Address Space and "Segments"	3-19
14	Illustrations of Segment Types	3-20
15	Segment Characteristics	3-23
16	Segment Types Overview	3-24
17	Table of Cray Origin2000 Segment Types ...	3-25
1	32-Bit Compatibility Areas	3-26
2	Addresses Accessed Based on CPU Mode	3-27
18	Cray Origin2000 Segment Types	3-28
19	Interpreting the Segment Type From the ...	3-29
1	User Address Area Segment	3-29
1	xkuseg - Virtual User Memory - mapped, ...	3-29
2	Kernel Address Area Segments	3-30
1	xkseg - Virtual Kernel Memory - mapped, ...	3-31
2	xkphys - Physical Kernel Memory	3-32
1	xkphys - unmapped, possibly CACHED	3-32
2	xkphys - unmapped, UNcached	3-32
20	The 64-bit Word and the Virtual Address	3-35
21	A Different View of Memory Segments - ...	3-37
22	"Unmapped" Virtual Address Segment Types	3-41
23	"Mapped" Virtual Address Segment Types	3-43
24	xkphys Memory Segments Diagram	3-45
25	xkphys Memory Segments - Detail	3-46
26	xkseg Memory Segment - Introductory ...	3-48
27	xkuseg Memory Segment - Introductory ...	3-50
28	xkuseg Memory Segment - Introduction	3-51
29	xkseg - Detail	3-53

1	xkseg Virtual-to-Physical Address ...	3-53
2	xkseg Virtual Addresses Mapped through ...	3-54
3	xkseg Wired Kernel TLB Entries - Diagram	3-56
4	xkseg Wired Kernel TLB Entries	3-57
30	Contents of xkseg Kernel Wired Entries	3-59
31	xkuseg - Detail	3-61
1	xkuseg "TLB Hit" - Diagram	3-62
32	Introduction to User Structures Related to ...	3-64
33	TLB Single Miss	3-66
34	Overview of Resolving a TLB Single Miss	3-68
35	Overview of Resolving a TLB Single Miss	3-69
36	Detail of Resolving a TLB Single Miss	3-71
37	Detail of Resolving a TLB Double Miss	3-72

Kernel Source Tree

4	Kernel Source Tree	4-1
1	Related On-Line Materials	4-2
2	Operating System Release Project Web ...	4-4
3	Source Code Location	4-5
4	Base Source Code Naming Convention ...	4-6
5	Where Is the Most Recent Version of the ...	4-7
6	Where Is the Most Recent Version of the ...	4-8
7	Summary: Location of Operating System ...	4-9
8	Kernel Source Tree Location	4-10
9	Kernel Source Tree Contents	4-11
1	The Difference Between ".h" and ".c" ...	4-12
2	Where to Find ".h" Files	4-13
10	Operating System and Kernel Source Tree ...	4-14
11	Tools Available to Browse Source	4-17
12	Determining What Software the System Is ...	4-18
1	versions - show system software; list ...	4-19
2	uname - show system software	4-20
13	How Do I Know What Crashed My System?	4-21
14	Where Does the System Put Things When ...	4-22
15	What System Logs Exist?	4-23

1 System Logs in /var/adm	4-23
2 Description of system logs, ...	4-24
Operating System Overview	
5 IRIX Operating System Overview	5-1
1 UNIX (IRIX) philosophy	5-2
2 IRIX system major components (user ...	5-3
3 IRIX system major components (kernel ...	5-4
4 When Does the Kernel Take Control Away ...	5-5
5 Kernel block diagram	5-6
6 Primary Kernel Activities	5-7
7 Summary of IRIX Kernel Primary Functions	5-10
Interrupt and Exceptions (Preliminary)	
6 Interrupts and Exceptions (Preliminary ...	6-1
1 Processor Operating Modes	6-2
2 Interrupt and Exception Types	6-3
3 How are Interrupts Different From ...	6-4
4 How are Interrupts Similar to ...	6-5
5 MIPS Processor Exception and Interrupt ...	6-6
6 General Exceptions	6-7
7 Hardware Interrupt Check	6-8
8 Software and Hardware Exception Check	6-9
Process Management Overview	
7 Process Management Overview	7-1
1 Process Management Overview	7-2
2 Executable Files and Processes Diagram	7-3
1 Executable Files and Processes Diagram	7-4
3 Executable Files and elfdump(1)	7-5
4 Process Definition Diagram	7-8
1 Process Definition Diagram Explanation	7-9
5 User Stack Diagram	7-10
1 User Stack Diagram Explanation	7-11

6 Kernel Stack Diagram	7-12
1 Kernel Stack Diagram Explanation	7-13
7 Processes and Kernel Threads	7-14
8 Displaying process memory (gmemusage(1))	7-15
1 Cray Origin2000 System Workload ...	7-16
2 IRIX Physical Memory gmemusage(1) ...	7-17
3 Process Physical Memory gmemusage(1) ...	7-18
9 Process Control Diagram	7-19
1 Process Control Diagram Explanation	7-20
10 Process Segments or Regions	7-21
11 Kernel's Region Tables Diagram	7-22
1 Kernel's Region Tables Diagram ...	7-23
12 Region Sharing Diagram	7-24
1 Region Sharing Diagram Explanation	7-25
13 Multiprocessing	7-26
14 Process Execution Flow Diagram	7-27
1 Process Execution Flow Diagram ...	7-28
15 System Call Interface Diagram	7-29
1 System Call Interface Diagram ...	7-30
IRIX System Calls	
8 IRIX System Call Processing	8-1
1 System Call Review	8-2
2 System Call Component Diagram	8-4
3 System Call Overview	8-5
4 System Call Walk Through	8-7
1 User Makes System Call	8-7
2 Sample assembler code for open(2)	8-8
3 Kernel Traps the Interrupt	8-9
4 Syscall() Dispatches the Call	8-10
5 Kernel Performs Specific System Call	8-11
6 Syscall() Resumes Processing	8-12
7 Systrap() Resumes Processing	8-13
8 User Resumes Processing	8-14
5 System Call Argument Processing	8-15

1	System Call Argument Processing	8-16
2	icrash(1M) Samples	8-18
1	Process uthread Display	8-18
2	uthread Detail	8-19
3	Trace of open(2) System Call	8-20
4	Trace Detail (partial)	8-21
5	Frame For open()	8-23
6	Disassembly Code For open()	8-24
7	Frame For copen()	8-25
8	Disassembly Code For kernel copen()	8-26
3	Register Aliases	8-27

Memory Management Overview

9	Memory Management Overview	9-1
1	Module Overview	9-2
2	Module Objectives	9-3
3	Hardware Memory Review	9-4
1	Origin2000 distributed-shared memory	9-5
2	Origin2000 Memory Hierarchy (in order ...	9-6
4	Hardware Address Sequence Review Diagram	9-7
1	Hardware Address Sequence Review ...	9-8
5	Memory Subsystem Introduction	9-10
6	Historical Solutions to Memory ...	9-11
7	Recent Solution to Memory Management ...	9-12
8	User Process Components Review	9-13
9	User Process Virtual Memory Image	9-14
10	User Process Virtual Addresses	9-15
11	Virtual to Physical Address Translation	9-16
12	Translation Lookaside Buffer (TLB)	9-17
13	Translation Lookaside Buffer (TLB) ...	9-18
14	TLB "Hits" and "Misses"	9-19
15	Virtual Addressing Summary	9-20
16	Demand Paging Overview	9-21
17	Demand Paging Page Load Procedure	9-22
18	Demand Paging Advantages and ...	9-23

19	Page Stealing	9-24
20	Page Stealing Page Selection	9-25
21	Page Stealing Page Actions	9-26
22	Page Stealing and Job Classes	9-27
23	Page Cache in IRIX	9-28
24	User Process Space and Swapping	9-29
25	Swap Space Management	9-30
26	The Swapper Process	9-31
27	The Swapper Process in IRIX	9-32
28	The Swapper Process Relationship to ...	9-33
29	Reporting Paging Activity (sar -p)	9-34
30	Reporting System Swapping and Switching ...	9-37
31	Reporting TLB Activity (sar -t)	9-39
32	Process Size (ps -l)	9-41
33	Reporting Memory Statistics (sar -R)	9-42
34	Reporting Unused Memory Pages and Disk ...	9-45
35	Reporting Memory Activity (gr_osview(1))	9-48

UNIX Filesystem Overview

10	UNIX Filesystem Overview	10-1
1	Sample UNIX FileSystem	10-2
2	Generic UNIX FileSystem	10-3
3	UNIX System V filesystem	10-4
1	Small UNIX file sample	10-5
2	Small UNIX file	10-6
3	Large UNIX file sample	10-7
4	Large UNIX file	10-8

XFS Filesystem - Structure

11	The Extent Filesystem (EFS)	11-1
11	xFS: the extension of EFS	11-2
11	A New XFS Filesystem	11-3
11	Allocation Group	11-4
11	Superblock	11-5
11	AGF: Free Space Block	11-6

11 A.G. Free Space List	11-7
11 AGFL - Allocation Group Free List	11-8
11 AGI: Inode Btree Control	11-9
11 AGI and Inode Btree	11-10
11 On-disk Inode	11-11
11 On-disk Inode (256 bytes) with local ...	11-12
11 I-block Directory	11-13
11 Btree Directory	11-14
11 Btree Directory - Index Block	11-15
11 Attribute Fork Inside Inode	11-16
11 Attributes Block	11-17
11 Data Fork - Binary Tree	11-18
11 Journaling Log	11-19
1 Sequence for replaying the log when the ...	11-20
11 I/O Performance	11-21
11 xfs_db printable block types	11-22
11 Mounted Filesystems	11-23

XFS File Management

12 File System Switch	12-1
12 XFS Code Architecture	12-2
12 Example IRIX read(2) Sequence	12-3
12 System Call Layer - Read	12-4
12 (XFS) Filesystem Layer - Read	12-5
12 System Buffers	12-6
1 detail on the vnode's page hash list	12-7
12 Example IRIX write(2) Sequence	12-8
12 (XFS) Filesystem Layer - Write	12-9

XFS File Management

13 Reference	13-1
1 mmap(2) - Memory Mapping a File	13-2

pfdatas

TR-IKI rev 0.7b SGI Proprietary

22jul1998

viii

14 Reference	14-1
1 pfdat's	14-2
2 Address Translation	14-3
3 Table locations	14-4

Disk I/O

15 Example IRIX read(2) Sequence	15-1
15 XLV Structure	15-2
15 XLV Driver Layer	15-3
15 ORIGIN module overview	15-4
15 Disk Device Connections to be Pictured ...	15-5
15 Hardware Graph format	15-6
15 Hwgraph Example	15-7
15 HWGraph Information Labels	15-8
15 Disk Driver Layer	15-9
15 SCSI Driver Layer	15-10
15 I/O Address Space	15-11
15 Interrupt Processing	15-12

IRIX Dumps - 6.5

16 IRIX Dumps - 6.5	16-1
1 Dump Scenario Diagram	16-2
2 Dump Scenarios	16-3
1 Four causes	16-3
2 Common code	16-5
3 Panic within panic	16-5
3 Processing Activities	16-6
1 Hardware Exception	16-6
2 NMI (Non-Maskable Interrupt)	16-8
3 Assertions	16-10
4 Panics	16-14
4 Common Routines	16-16
1 cmn_err() Panic Processing	16-16
2 icmn_err() Panic Processing	16-17
3 syncreboot() Processing	16-19

TR-IKI rev 0.7b SGI Proprietary

22jul1998

ix

4	dumpsys() Processing	16-20
5	dumpvmcore() Processing	16-21
5	Dump Level Configuration	16-23
1	Dump Level	16-23
2	Dump level meanings	16-24

Bibliography

17	Bibliography	17-1
1	BOOKS BY SGI EMPLOYEES (former or ...)	17-2
2	BOOKS	17-3
3	ON-LINE DOCUMENTS	17-4
4	TOOLS	17-5
5	TRAINING MATERIALS WEB PAGES	17-6
6	INSTRUCTOR WEB PAGES (links to their ...)	17-7
7	ENGINEER WEB PAGES	17-8

Appendix A: Origin2000 Support ...

A	Origin2000 Support Processes For High ...	Appendix A-1
1	Purpose	A-2
2	Getting Help	A-3
3	Getting Cray domain accounts	A-4
4	Site Planning	A-5
5	Installation Planning	A-6
6	System Registration	A-7
1	System Serial Number	A-8
7	Installation Reporting	A-9
1	Initial Mainframe Hardware Install ...	A-10
2	Initial Mainframe Software Install ...	A-11
3	Hardware & Software Installation Defects	A-12
8	System Failure Reporting	A-13
9	Problem Escalation	A-14
1	GTS's Hodlist	A-15
2	U.S. Escalation Model	A-16
3	International Escalation Model	A-17
10	Problem Reporting	A-18

x

22jul1998

TR-IKI rev 0.7b SGI Proprietary

1	Software Problem Reporting	A-19
11	Customer Communication	A-20
1	Pipeline	A-21
2	Cray Inform (CRInform)	A-22
3	Field Notices and FYIs/FIBs/NPIs	A-23
12	Related Information	A-24

Appendix B: CPU R10000 Overview

B	MIPS ® R10000 Microprocessor Overview	Appendix B-1
1	Instruction prefetch	B-2
2	Out-of-order execution	B-3
3	Queuing structures	B-4
4	Integer Queue	B-5
5	Floating Point Queue	B-6
6	Address Queue	B-7
7	Execution Units	B-8
8	Integer ALUs	B-9
9	Floating-Point units	B-10
10	Load/Store unit and the TLB	B-11
11	Secondary Cache Controller	B-12
12	System Interface	B-13
13	R10000 Branch Unit	B-14
1	Branch instruction problem	B-15
2	Branch prediction	B-16

Appendix C: region.c source file

C	region.c source file (excerpt)	Appendix C-1
---	--------------------------------	--------------

Appendix D: sbd.h

Appendix E: kldir.h header file - has ...

E	kldir.h header file (excerpt)	Appendix E-1
---	-------------------------------	--------------

xi

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Appendix F: IRIX 6.5 Kernel Values

F IRIX 6.5 Kernel Values

- 1 Kernel Value Table
- 2 Column Meanings
- 3 Kernel Value Table
- 4 Sample "kerninfo" output
 - 1 Live Indy Workstation (IRIX 6.5 beta)
 - 2 O2000 system dump (IRIX 6.5 beta)
 - 3 O2000 live system (flurry; IRIX 6.5 ...

Appendix F-1
F-2
F-3
F-4
F-5
F-6
F-7
F-8

Appendix G: How to get a core dump from ...

G How to get a core dump from your Indy ...

- Figure A-0: Critical Problem Escalation
- Figure A-1: Cray Origin 2000 U.S. ...
- Figure A-2: Cray Origin 2000 U.S. Field's ...
- Figure A-3: Cray Origin 2000 International ...

- Table 3-0: Segment Types and Characteristics for ...
- Table A-1: Site Planning Materials

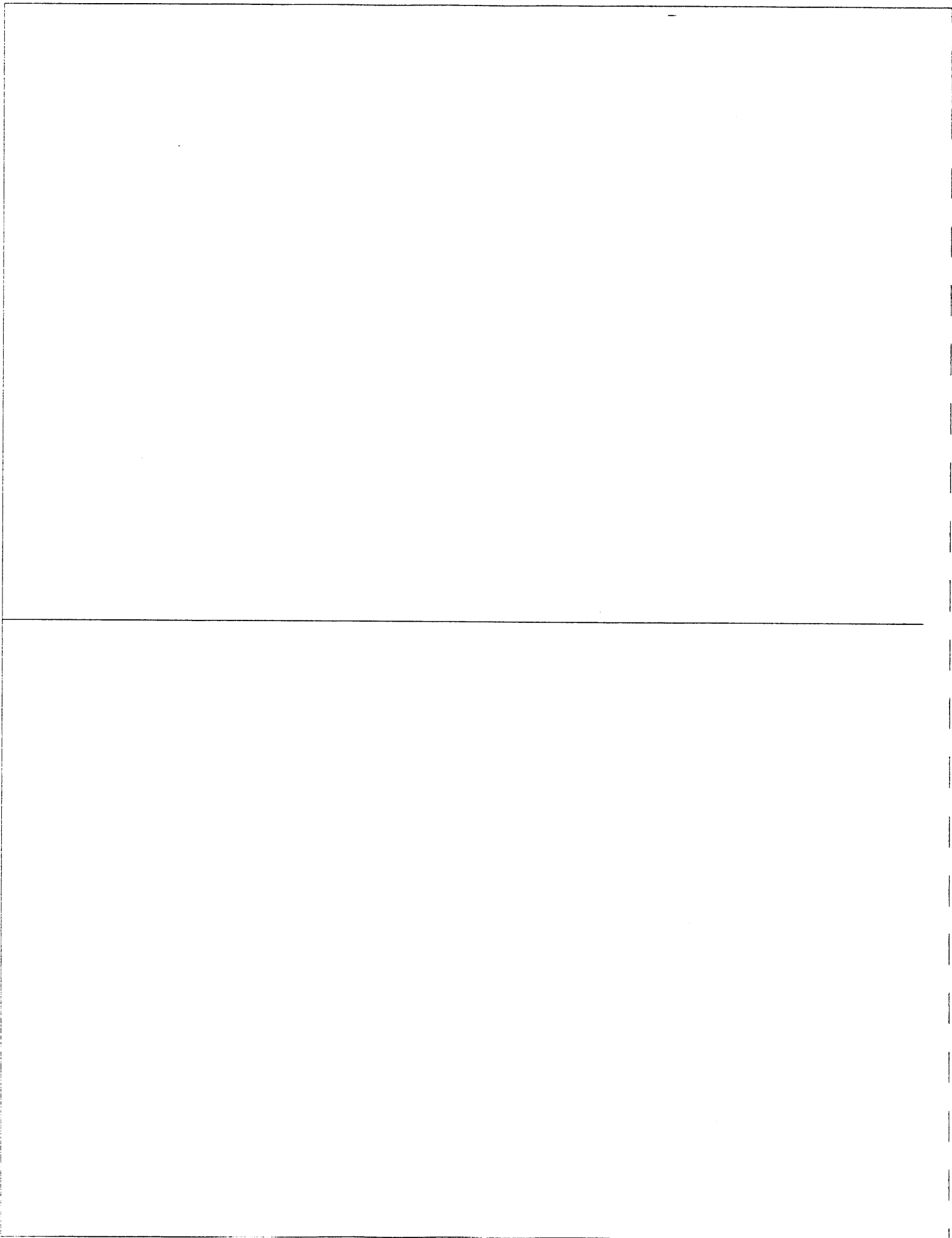
Figures

Appendix G-1

A-14
A-16
A-16
A-17

Tables

3-25
A-5





IKI: IRIX Kernel Internals Home Page



IKI65: IRIX 6.5 Kernel Internals

CRAY
PRIVATE

Training Materials

- SPT course description
- Day 1: Introductory Lessons (with separate or merged TOC window)
 - IRIX source browsing (`cscope(1)` and `dwarfdump(1)`) 
 - Introduction to Dump Analysis (showcase) (html) (*Matt Robinson*) 
 - `icrash(1M)` Tutorial *Draft 1.0*
 - Dump Analysis *Draft 1.0*
- Day 2: Lessons (with separate or merged TOC window)
 - Process Memory Study (lab 0)
 - User Virtual Address Study (lab 1)
- Day 3: File System Lessons (with separate or merged TOC window)
 - filesystem structure
 - file management
- Day 4: Input/Output Lessons (with separate or merged TOC window)
 - I/O layer
- Day 5: Lessons (with separate or merged TOC window)
 - Dump Analysis

1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Training Material Utilities

- Request CrayRealm and Training domain accounts

 Search training materials:



 Search training glossary:

2

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Module 1: IRIX Software Training



IRIX Software Training

CRAY
PRIVATE

Request CrayRealm and Training domain accounts

Contents

1. Class Materials
2. Reference Materials
3. Cellular IRIX
4. Mail & Newsgroups
5. Performance
6. Performance Co-Pilot (PCP)
7. Application Programming
8. Hardware Reference Materials
9. Other Reference Materials

 Search training materials:



 Search training glossary:


1-1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Class Materials (SGI Employee Use Only)

I65RU: *IRIX 6.5 Release Update*

SPT course description & class materials (HTML) 

IKI65: *IRIX Kernel Internals (IRIX 6.5/Kudzu)*

SPT course description & class materials (HTML & PostScript) 

OPET: *O2000 Performance Evaluation And Tuning*

SPT course description & class materials

PESTO: *Performance Evaluation and System Tuning for Origin2000 and Onyx2*

Customer Education course description & class materials (same as OPET)

IFO: *IRIX Functional Overview* 

Customer Education Project Plan (Working Draft)

- Hardware & IRIX Operating System Overviews (Working Draft)
- File, I/O, Memory & Process Management (Working Draft)
- Interprocess Communication (Working Draft)
- Security Features (Working Draft)
- Data Migration Facility (DMF) (Working Draft)
- Domain Name Service (DNS) (Working Draft)
- Network File System (NFS) (Working Draft)
- Network Information Service (NIS) (Working Draft)
- TCP/IP (Working Draft)
- Unified Name Service (UNS) (Working Draft)

1-2

22jul1998

TR-IKI rev 0.7b SGI Proprietary




Reference Materials

- Origin 2000 Support Processes and Tools (lesson) | (slides only: PostScript | Showcase)
- Origin 2000 picture, Cray Origin 2000 picture, news items, and hardware Options/Enhancements milestones *CrayReal™*
- IRIX Source & Object code location and descriptions (lesson)
- Advanced Systems Division's Home for High Performance Computing

An internal SGI resource supporting the technical marketing and development of high-end compute products.

- HPCxchange newsletter at the new HPC Web Site
- Silicon Sales: Hardware, Software, Services & Support
- ASD Marketing System Administration Team provides HW and SW support for ASD's Technical Compute Division and the Graphics Division.
- Origin2000 training material and presentation slides
- ORIGIN-LINKS
- Origin Benchmark Resources (MV & Eagan)
- How to reconfigure an Origin 2000 to 180MHz 1MB Cache system



Projects & Products	Irix 6.4 - Ficus SPR Query	 Irix 6.5 - Kudzu Features by Number, Category Exceptioned Features, SPR Query
	ProjectVision (PV) viewer and process BugWorks: Web PV database interface bwx: command line PV database interface	
	PatchWorks: patch database interface (for IRIX 6.4.1) Patch Process FAQ, patch types, tools, colors, PV+ Tool, browsers	
InfoWorks	Software Development & Release Information	
SGI University	Software Engineering slides and videotapes	

- SGI's Top, Kudzu's, *comp.sys.sgi's*, and Matthias Fouquet-Lapar's Frequently Asked Questions (FAQs)
- Origin 128 detailed issues and problem information and individual responsibilities CRAY REALM
- To obtain *Uncle Art's Big Book of IRIX*:

1. telnet dist.engr as user guest and no password
2. cd /sgi/doc/swdev/BigIrixBook
3. ftp the postscript (*.out) files from that location.

Alternately, (may not work for you):

1. Install the handbook utilities: inst -f dist.engr:/sgi/infotools
2. Enter: handbook -s dist.engr:/sgi/doc/swdev/BigIrixBook

- Automating IRIX 6.5 Miniroot Installs with RoboInst

1-5

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Legend: ● = Very useful, ● = Somewhat useful, ● = Not rated

- IRIX Kernel Development Information Index (*John Hesterberg & Tom Cox*)

Information Index of IRIX Kernel Development Process at Cray; How to set up accounts, scan source code using `cscope(8)` and other tools. Has details down to building kernels and testing your code.

1-6





22jul1998

TR-IKI rev 0.7b SGI Proprietary

Keyword search:			
Content for:	IRIX 6.2	IRIX 6.3	IRIX 6.5
Select an item below to restrict your search:			
<input type="checkbox"/>	Online Books	<input type="checkbox"/>	Man Pages
<input type="checkbox"/>		<input type="checkbox"/>	Release Notes

● Tech Digest links to many useful items for Field Analysts

TECH.Digest:	Overview	Collections	FAQ	Search	Feedback	Internal: SilJunc Sniff Int.Sites		
Views:	Home Comm Gfx HA Hdw Lang MMed Unix					External: Sil.Surf TechCtr FTP		
Bugs	Bulletins	CMSInfo	CSE.Lab	DTbox	FTPInfo	HWDevBk	InstLoc	License
Matrices	ManPages	Oasis	Parts	Patches	Pipeline	PriceBook	PubDomSW	QNADocs
RelNotes	Security	STbox	TechMail	TechPubs	--	Videos	FAQS: Top SWEngr	

Universal manpage:	
Top SGI FAQ:	
 Oasis	SGI Technical Support Information: Easy way to search bugs, calls, source, etc.
 Technical Publications	 

The IRIX section of the On-Line Technical Publications Library is an important resource for obtaining information about IRIX as well as all SGI software products. It has books for developers, system administrators, and end users, as well as a collection of books related to SGI hardware.

- **IS Origin2000 System Info**: descriptions of available O2000s systems and other information
- *Understanding and debugging Problems on the Origin2000 and Onyx2 Systems*
- *IRIX Device Driver Programmer's Guide (007-0911-060)*

Documents the execution environment for kernel-level and user-level device drivers in IRIX. Covers development tools and methods used to create device drivers.

- Dean Roehrich's SGI/IRIX Testing GrabBag includes how to install Ficus on Drive 2 of your Indy



Internal Support Tools

(IST) Group products, distribution center, and information center

Tool	Description	User Guide	Training Slides	Reference Manual
AvailMon	Availability Monitor	(PS PDF)	(ShowCase)	(PDF)
FRU	Field Replaceable Unit Analyzer	(PS PDF)	(ShowCase)	(PDF)
ICRASH	Irix Crash Analyzer	(PS PDF)	(ShowCase)	(PDF)
IPM	Installation Planning Manager			
MDK	Micro-diagnostic kernel	(PS PDF)		
POD	Origin2000 Power-on Diagnostics			(PDF)
RAT	Remote Access Tool	(PS PDF)	(ShowCase)	(PDF)
SVP	System Verification Program	(postscript)	(ShowCase)	(PDF)

NOTE: The PDF format is readable by the Adobe Acrobat Reader. If you do not have a copy, please download the latest version.

• Other Internal Support Tools:

Diagnostic Roadmap for Origin2000 and Onyx2 (preliminary)
What Tool to use, and when (general)
Pre-installation / Upgrade Support
Installation / Upgrade Support
Repair Support
Preventive Support

Cellular IRIX



- *Cellular IRIX Plans; G. Broner (30jan97) and time estimates (27jan97)*

Discusses Cellular IRIX's overall long-term design direction. Describes intermediate deliverables needed to meet Enterprise and HPC computer market short term needs.

- Cellular IRIX Documentation Navigator

- Common Operating System Plan for SN1 (28Jun96)

Discusses OS direction for SN1 and transition from SN0 and T3E in support of SN1 Common OS plan.

- Cellular IRIX and Nexus OS Infrastructure

- Presentations
- Nexus Architecture and Infrastructure
- Cellular IRIX Subsystems

- Cellular IRIX Related Lego Design Documents

- Cellular IRIX Project Planning and Product Specification page (03feb97)

- *Cellular IRIX 6.4 Technical Report*

Mail & Newsgroups



MAILMAN: web interface to the Majordomo mailing list manager. 

Newsgroup	Name	Actions
SNO Applications/Performance	snappl.engr.sgi.com	Subscribe Unsubscribe
SNO OS	sn0.csd.sgi.com	Subscribe Post Unsubscribe
TechMail Archives	see TechMail Overview	Sort Archive by Group/Month Search Archive by string
CPS Majordomo News Groups	Quick Reference	List all List subscribed

Performance

- Availability Monitor (AvailMon and IRS Audit/KPM) home page and training slides
- Miser: User level program that generates a non-conflicting schedule of jobs with known time and space requirements.
- OPET: O2000 Performance Evaluation And Tuning class
- *Origin2000/Origin200/Onyx2 Quick Reference Single-Processor Tuning* (PostScript | PDF)
- Origin 2000 Performance Report (20May97: postscript, html, frame) and slides
- Origin 200 Performance Report (17mar97: (postscript, html, frame)
- Performance analysis tools
 - Process Activity Reporter (par) (par for dummies)
 - kernel function profiling (prfpr)
 - System monitoring: `gr_osview(1)`, `sar(1)`, `osview(1)`
- *Performance Tuning Optimization for Origin2000 and Onyx2* (007-3430-001)
(summary | manual | glossary)

Performance Co-Pilot (PCP)

PCP provides a range of services designed to help monitor and manage system performance.

- Origin Topology and Monitoring
- PCP engineering and marketing home pages
- PCP is developed and maintained within the EBU Performance Tools Group (PTG)
 - PTG's projects and future PCP product releases.
- Installation, Licenses and Test Drive Instructions

1-17

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Application Programming

- Compiler Group:
 - Mongoose Compiler 7.2 Project
 - Project Caribou
 - Compiler-related environment variables
 - Irix 6.5 pthreads
- CPS: Support Planning Operations (SPO) 7.2 Compiler status
 - Schedule and Dependencies, Features, Pre-Release Test Program, Customer Impact, Manufacturing Release (MR) Status
- SGI/CRAY Supercomputing Application Programming Interface (API) (004-2211-001: HTML | other)
 - Describes supercomputing API. Defines programming environment elements including compilers, language libraries, system libraries, environment variables, system calls, and a few associated utilities.
- PIPELINE ARTICLE 19960405: Programming Tips for IRIX
 - Contains information on useful IRIX system utilities, programming interfaces, and large memory allocation troubleshooting techniques that may be of value to developers writing IRIX applications.
- *Topics and IRIX Programming* manual
- *Programming on Silicon Graphics Computer Systems: An Overview* manual
- *Power Fortran Accelerator User's Guide* manual
- *Origin and Onyx2 Programmer's Reference Manual* (007-3410-001)
 - Describes memory maps, and physical and virtual address spaces; including Hub Special, I/O, Memory Special, and Uncached spaces.

1-18

22jul1998

TR-IKI rev 0.7b SGI Proprietary

● NCSA & Boston University's Silicon Graphics Origin2000 Supercomputer Repository

NCSA and Boston U jointly announced this web site at the CUG Origin 2000 in Minneapolis Oct '97. It's intended to be a public repository for Origin2000 information, and a catalyst for discussion.

[Home](#)

[Links](#)

[Benchmarks](#)

[Jobs](#)

- [Links to National Computational Science Alliance and other Origin2000 Sites](#)
- [Partial List of Origin 2000 Scientific Applications \(Scalability & Performance charts\)](#)
- [LANL's Preliminary Performance Study of the SGI Origin2000](#)
- [Performance of Fortran 90 Array Intrinsic Functions on the SGI Origin2000](#)

Hardware Reference Materials

- High End Engineering MFG Test Engineering pages contain very useful HW reference materials.

- *Info Tools for High End Production*

- *Technical Overview of the Origin Family*

- Introduction
- Origin2000 Components
- What Makes Origin2000 Different
- Scalability and Modularity
- Systems Interconnections
- Crossbar
- Distributed Shared Address Space(Memory and I/O)
- System Bandwidth

- CRAY Origin 2000 64 Processor Beta Information ~~2000~~
- Origin 2000 system part numbers, descriptions, and quantities
- Lego Design Document Index
- USFO Sales Tools listed and documented
- Origin & Onyx2 World-Wide Service Support Tools project page and tool descriptions
- Remote Access Tool (RAT) User's Guide for curses tool that talks to the System Controllers.
- *Origin & Onyx2 Theory of Operations Manual: (007-3439-001)*

TOC, architecture overview, boards, ASICs, glossary

- *IP27prom Technical Reference Manual*

Covers usage of the IP27prom to boot or debug an Origin2000 system:

- Module System Controller (MSC) including commands
MSC was formerly known as ELSC (Entry-Level System Controller)
- Multi-Module System Controller (MMSC) including debug switches
MMSC was formerly known as FFSC (Full-Featured System Controller)
- CrayLink Interconnect Topology Primer

- IP27prom Operation including booting
- IP27prom Command Set
- IP27prom debugging including LED error codes and log messages

Includes sufficient background Origin2000 information to minimize the number of documents required to use the IP27prom.

- Multi-Module System Controller (MMSC)
 - commands, security, flashing MMSC Firmware
 - Flashing: reloading a PROM's firmware image.*
- MIPS R10000 Superscalar Microprocessor
- MIPS Programming Manuals:
 - *MIPSpro Compiling and Performance Tuning Guide*
 - *MIPSpro 64-bit Porting and Transition Guide*
 - *MIPSpro Assembly Language Programmer's Guide*
 - *MIPSpro N32 ABI Handbook*

1-20.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Other Reference Materials

- CUG Papers, San Jose, May 1997:
 - *Examples of Various Approaches to High-Performance Computing through Scalable Systems* CrayRealin
 - *A Comparison of Application Performance Across Cray Product Lines* CrayRealin
- CPS: Support Planning Operations
 - for O2, Octane, Onyx2, Origin 200, SGI & Cray Origin 2000, TPU, IRIX 6.5
 - Project status, Planning documentation & minutes, Service Readiness Review (SRR), Service requirements, and systems
 - IRIX 6.5 Support Readiness Information
 - Chandler Lai - Support Planning
 - Presentations done by engineering groups which are planned to be available as videos on demand from servinfo
 - CPS IRIX 6.5 (Kudzu) Project
 - SSE System Administration class
 - SSE Network Administration class
 - IRIX 6.5 New Features and Differences class
- Server Central: Origin 2000 & IRIX Product Information
 - Advanced Server & Workstation Environments Product MR Status
 - Trusted Irix, IRIX 6.2| 6.3| 6.4| 6.5, XFS, DMF DCE/DFS
 - Origin 2000 Configuration Guide (PS | PDF), Data Sheet, & Product Guide
 - Cray Origin 2000 System Descriptions (PDF),
- Cray Software Engineering Technical Forums: previous and planned
- CRAY Scalable Node and Origin 2000 project home pages and their list of SN-related links.
- IRIX 6.5 Public Technology Focus & Roadmap & Archives
- Kudzu Early Access Delivery List by *Linda Conroy*
- Kudzu 128P test plan by *Bill Roske*
- Silicon Sales: Presentations on Demand (POD) Origin 2000 Presentation Overview & Features

1-21

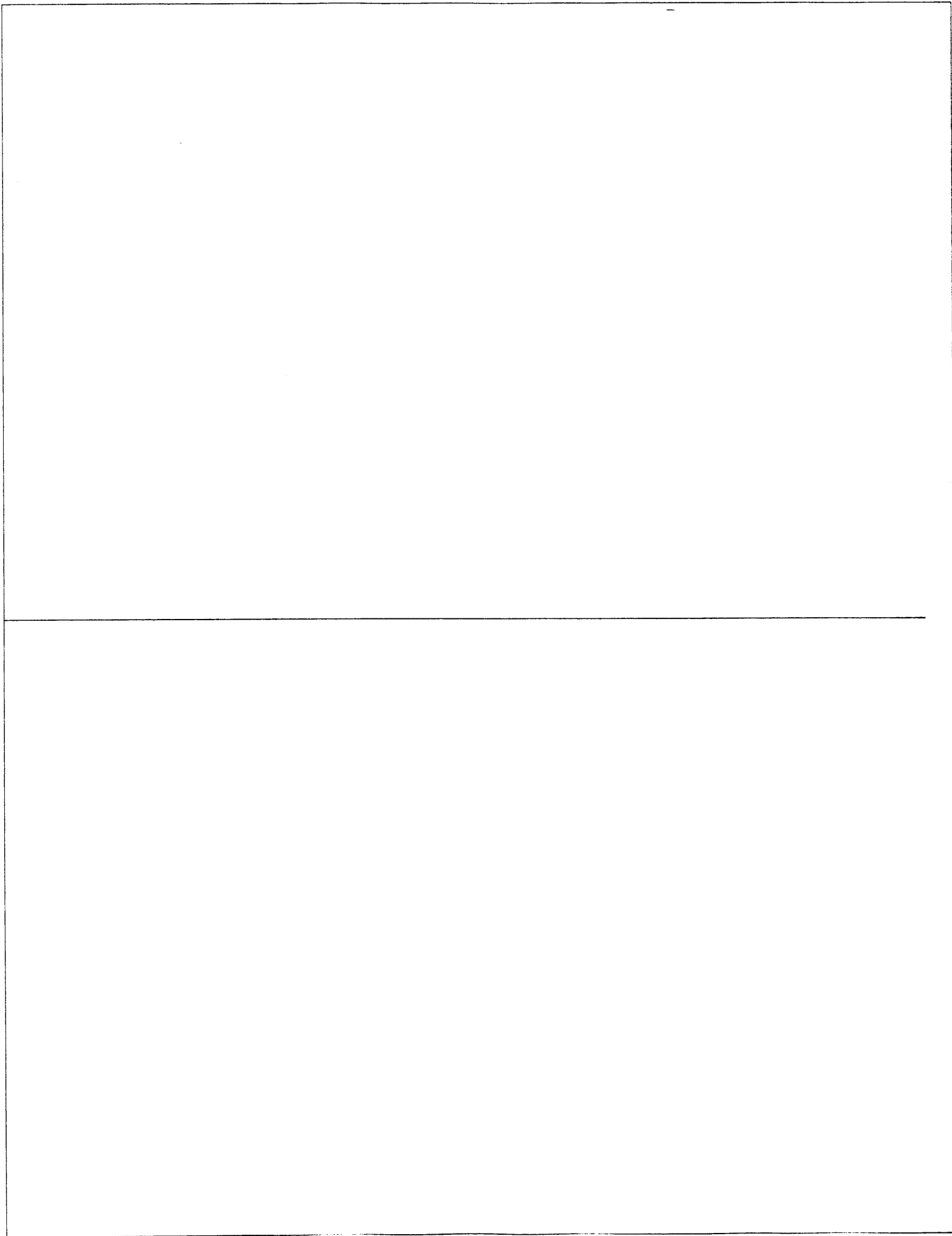
22jul1998

TR-IKI rev 0.7b SGI Proprietary

- Technical resources referenced in an SGI Pipeline article on Cellular Irix
- *The Magic Garden Explained* by Bernard Goodheart and James Cox

Authoritative, in-depth description of internal working and programmatic interfaces to UNIX System V Release 4 OS. Explains various techniques, algorithms, and structures within UNIX SVR4 kernel.

- Wind River Systems VxWorks R/T OS



Module 2: Cray Origin2000 Architecture

Cray Origin2000 Architecture

2-1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Cray Origin2000 Architecture Module Overview

This section provides a hardware overview of the Cray Origin2000 architecture.

By the end of this section, the student should be able to describe each of the below:

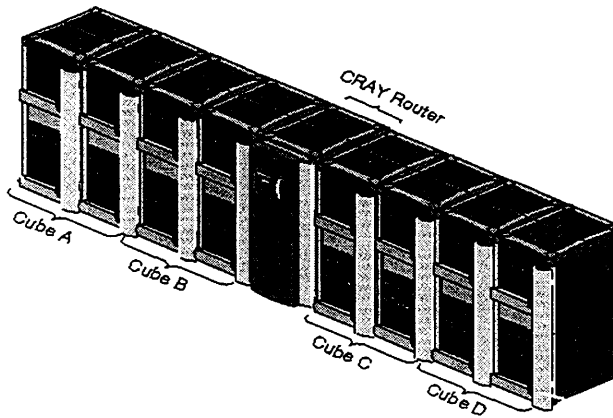
- Hypercube Structure
- Module Architecture
- Router Connections
- Node Board, XBOX, and Router Relationship
- R10000 Chip Architecture
- Cache Memory Systems
- Non-Blocking Cache
- Cray Origin2000 Cache Types

2-2

22jul1998

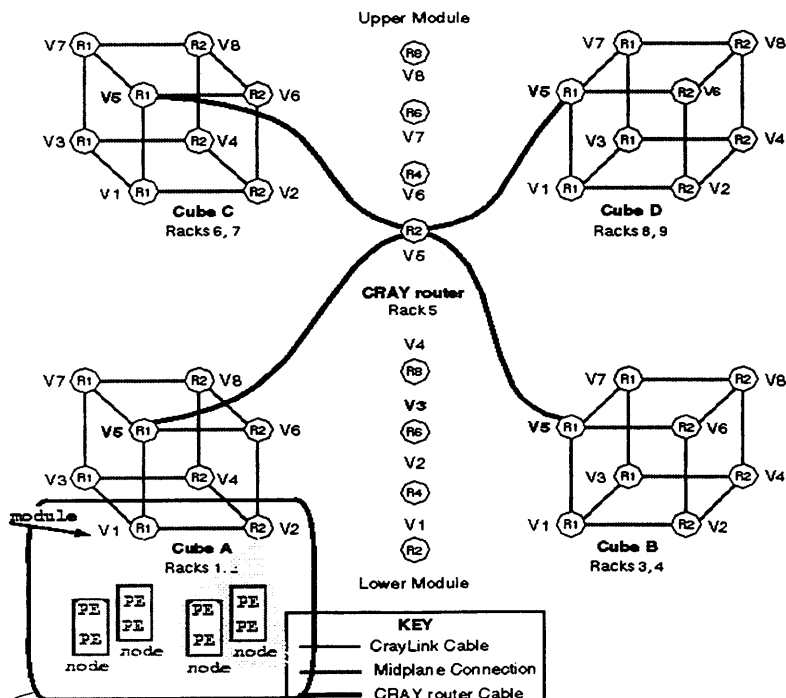
TR-IKI rev 0.7b SGI Proprietary

CRAY Origin2000 Multirack System



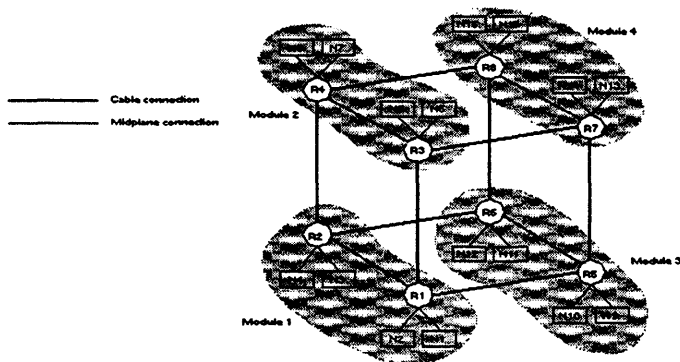
The CRAY Origin2000 system is a multirack system that can interconnect up to a maximum of 128 CPUs in 9 racks (8 server racks and 1 CRAY router rack) that are arranged as 4 cubes of 32 processors each. The router rack interconnects the processors with CrayLink cables.

Router and Hypercube Connection



The above diagram presents a conceptual view of the nine rack, four cube, 128 CPU system pictured in the previous illustration. Each "hypercube" has a variety of pathways to connect router vertices (V1, V2, etc.). A "module" is made up of a pair of routers (R1 and R2). Each module is made up of two node boards. Each node board (usually) contains two CPU's (Central Processing Units), also known as PE's (Processing Elements).

Hypercube

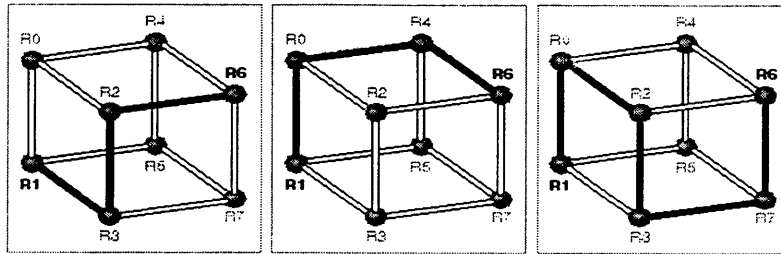


The above diagram focuses on a single hypercube and its router connections, as well as the module configuration. Two router vertices and their four nodes are considered a module. Every module contains eight CPU's.

Each of the eight router vertices connects to two nodes. Since each node contains two CPU's, a single hypercube contains:
 $(8 \times 2 \times 2) = 32$ CPU's.

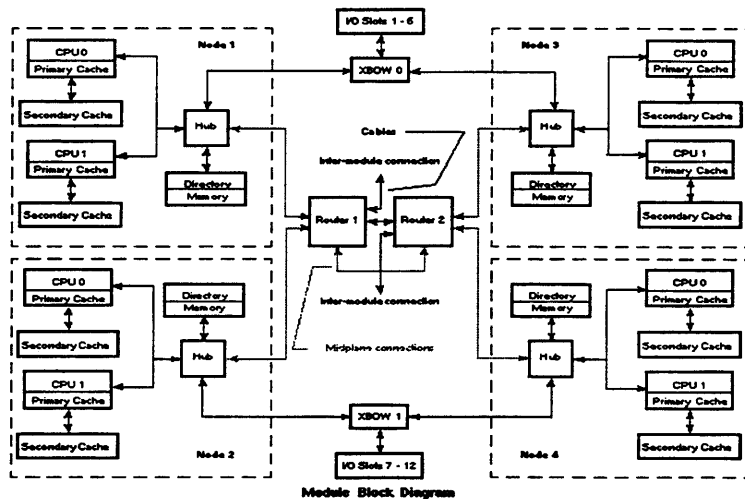
A four hypercube configuration contains 128 processors (4×32) .

Origin2000 redundant paths



The interconnection fabric provides a minimum of two separate paths to every pair of Origin2000 nodes (and their total of four CPU's). The above diagram illustrates three different paths from node R1 to node R6. This redundancy allows the system to bypass failing routers or broken interconnection fabric links. Each fabric link is additionally protected by a CRC code and a link-level protocol, which retry any corrupted transmissions and provide fault tolerance for transient errors.

Module and Node Block Diagram



There are four nodes in a module. Each node contains two CPU's, for a total of eight processors per module.

Each Origin2000 node board is a "system on a board". The Origin2000 has a number of processing nodes linked together by an interconnection fabric. Each processing node contains:

- 1-2 R10000 processors

- A portion of shared memory (64 MB to 4 GB)
- A directory for cache coherence
- Two interfaces:
 - a connection to I/O devices
 - a connection of all the system nodes through the interconnection fabric.

Node Board Components

The Origin2000 central node board can be viewed as a system controller from which all other system components radiate. Primary Origin2000 components are as follows:

- Processor

Origin2000 system uses the MIPS[®] R10000, a high-performance 64-bit superscalar processor which supports dynamic scheduling. Some of the important attributes of the R10000 are its large memory address space, together with a capacity for heavy overlapping of memory transactions (up to twelve per processor in Origin2000).

- Memory

Each node board added to Origin2000 is another independent bank of memory, and each bank is capable of supporting up to 4 GB of memory. Up to 64 nodes can be configured in a system, which implies a maximum memory capacity of 256 GB.

- I/O Controllers

Origin2000 supports a number of high-speed I/O interfaces, including Fast, Wide SCSI, Fibrechannel, 100BASE-Tx, ATM, and HIPPI-Serial. Internally, these controllers are added through XIO cards, which have an embedded PCI-32 or PCI-64 bus. Origin2000 I/O performance is added one bus at a time.

- CrayLink Interconnect

This is a collection of very high speed links and routers that is responsible for tying together the set of hubs that make up the system. The important attributes of CrayLink Interconnect are its low latency, scalable bandwidth, modularity, and fault tolerance.

- XIO and Crossbow (XBOW)

These are the internal I/O interfaces originating in each Hub and terminating on the targeted I/O controller. XIO uses the same physical link technology as CrayLink Interconnect, but uses a protocol optimized for I/O traffic. The Crossbow ASIC is a crossbar routing chip responsible for connecting two nodes to up to six I/O controllers.

- Hub

This ASIC is the distributed shared-memory controller. It is responsible for providing all of the processors and I/O devices a transparent access to all of distributed memory in a cache-coherent manner.

- Directory Memory

This supplementary memory is controlled by the Hub. The directory on each node keeps information about the cache status of its assigned subset of physical memory.

For every physical page in a node's local memory, there is a bit which indicates whether that page is in any processor's primary instruction cache, primary data cache, or secondary cache, and whether the data is "dirty" (needs to be written to disk) or "clean" (an unchanged copy of disk data).

This status information is used to provide scalable cache coherence, and to migrate data to a node that accesses it more frequently than the present node.

On architectures with the capacity for 32 or less CPU's, the directory memory uses part of the local memory assigned to the node. On architectures with the capacity for 33 or more CPU's, the directory memory is on a separate "Dual In-Line Memory Module", or "DIMM" plugged into the node board.

The main memory DIMM holds 16 bits of directory memory. The extended directory DIMMs hold an additional 32 bits of directory memory.

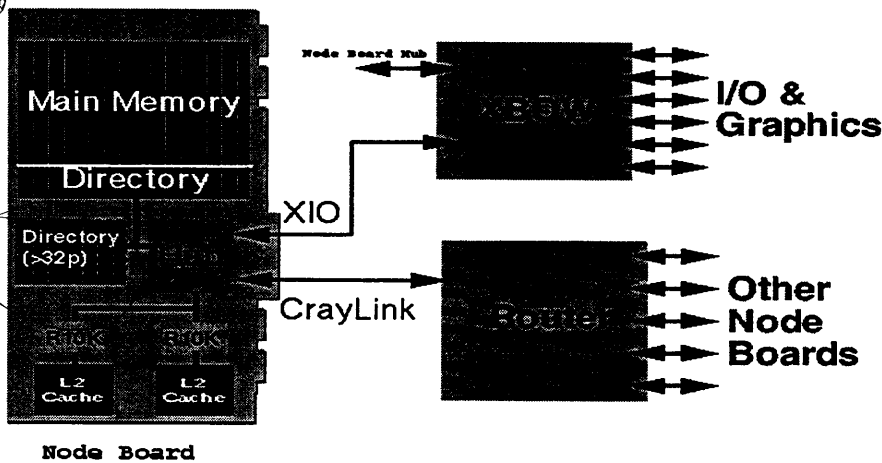
Each block of memory has a directory table that indicates the cached state of the block:

- Unowned: (uncached) the memory block is not cached anywhere in the system
- Exclusive: only one readable/writable copy exists in the system Shared: zero or more read-only copies of the memory block may exist in the system. Bit vectors point to any cached location(s) of the memory block.
- Busy states: Busy Shared, Busy Exclusive, Wait. These three transient states handle situations in which multiple requests are pending for a given memory location.
- Poisoned: page has been migrated to another node. Any access to the directory entry causes a bus error, indicating the virtual-to-physical address translation in the TLB must be updated.

The Hub ASIC is responsible for determining the state of the memory page during any memory request. The protocol is implemented completely in hardware.

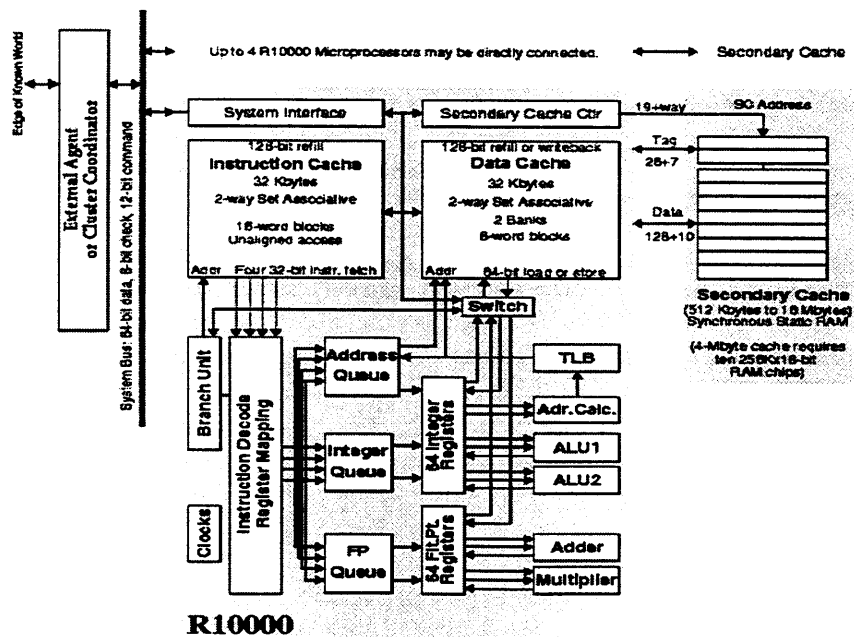
Node Board, XBOW, and Router Relationship

Separate chips DIMM on 732P machines



The Hub, XBOW, and Router are ASICs (Application-Specific Integrated Circuit) which act as switches to provide the interconnectivity of the Origin2000 components. The Hub is responsible for providing all of the processors and I/O devices a transparent access to all of the Origin2000 distributed memory. The XBOW (Crossbow) is responsible for connecting two nodes to up to six I/O controllers. The Router is responsible for connecting a pair of nodes to other node boards on the system.

MIPS[®] R10000 Microprocessor (block diagram)



The R10000 Microprocessor implements the MIPS[®] IV instruction set architecture. The R10000 Microprocessor delivers performance of 800 MIPS at a frequency of 200 MHz, with a peak data transfer rate of 3.2 GBytes/second to secondary cache.

More About the[®] R10000 Microprocessor

Instruction prefetch

- Out-of-order execution
- Queuing structures
- Integer Queue
- Floating Point Queue
- Address Queue
- Execution Units
 - Integer ALUs
 - Floating-Point units
 - Load/Store unit and the TLB
- Secondary Cache Controller
- System Interface
- R10000 Branch Unit
 - Branch Instruction Problem
 - Branch Prediction

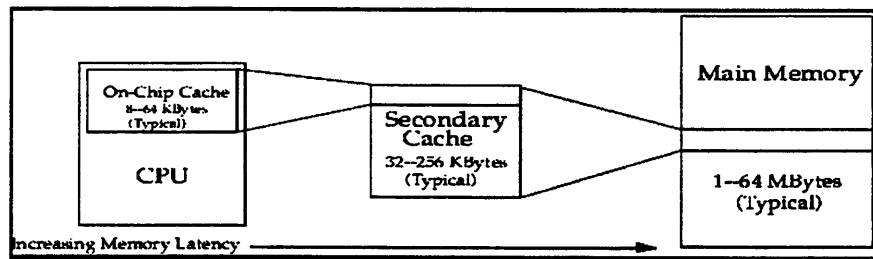
The contents of the above links can be found in the appendix : "MIPS[®] R10000 Microprocessor Overview".

More About Memory

Cache memory systems

A cache memory system is comprised of a small amount of memory which contains a block of memory addresses comprising a small section of main memory. Cache memory has much faster access times and can deliver data to the processor at a much higher rate than main memory.

On-chip cache memory systems can greatly improve processor performance because they allow accesses to be completed often times in one cycle. On-chip cache contains a range of addresses which comprise a subset of those addresses in the secondary cache. In turn, the secondary cache contains a range of addresses which comprise a subset of those addresses in main memory.



The above picture should be correct to show a secondary cache size range of from 512 Kbytes to 16 Mbytes.

Origin2000 Distributed Shared-Memory (DSM) and I/O

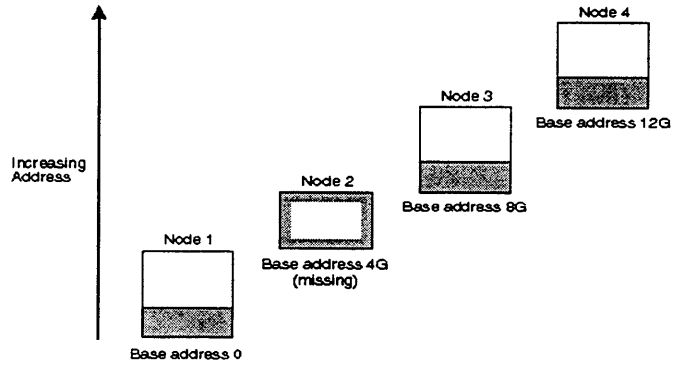
Origin2000 memory is located in a single shared address space but is physically dispersed throughout the system for faster processor access over the interconnection fabric. This differs from former systems, in which memory is centrally located on and only accessible over a single shared bus.

Page migration hardware moves data into memory closer to a processor that frequently uses it. This page migration scheme reduces *memory latency*- the time it takes to retrieve data from memory. Although main memory is distributed, it is universally accessible and shared between all the processors in the system.

Similarly, I/O devices are distributed among the nodes, and each device is accessible to every processor in the system.

The Origin2000 divides main memory into two classes: local and remote. Memory on, or assigned to the same node as the processor is labeled local, with all other memory in the system labeled remote. Despite this distribution, all memory remains globally addressable.

To a processor, main memory appears as a single addressable space containing many blocks, or pages. Each node is allotted a static portion of the address space. This means there is a gap if a node is removed. The illustration below shows an address space in which each node is allocated 4 GB of address space, and Node 2 is removed, leaving a hole from address space 4G to 8G.

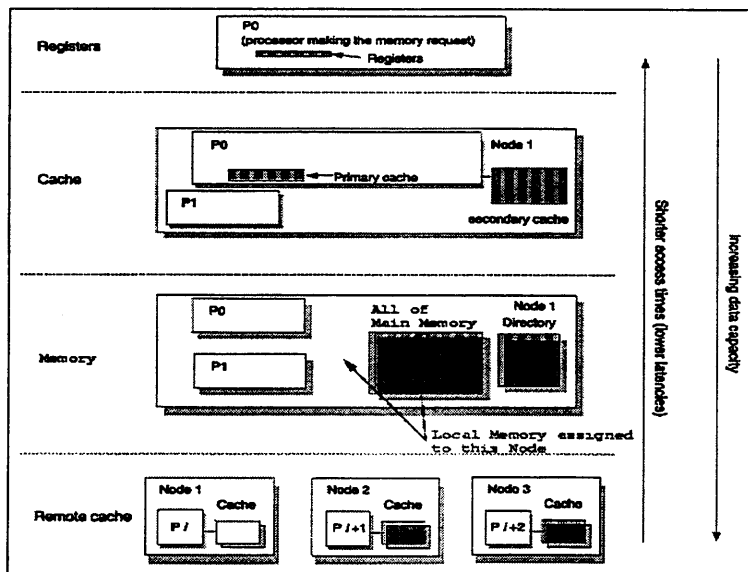


Physical address
address

NODE	BANK	PAGE	OFFSET
node	20mms		byte

2,048 (30)

Origin2000 Memory Hierarchy Diagram



Origin2000 Memory Hierarchy Explanation

Memory in Origin2000 systems is organized into the following hierarchy:

- Processor Registers
- Local Caches
- Memory
- Remote Caches

Processor Registers

The registers are closest to the processor making the memory request, which is the processor labeled P0 in the diagram. Since registers are physically on the chip they have the lowest latency, that is, they have the fastest access times.

Local Caches

The primary and secondary caches located on P0 are shown above (processor P1 has identical architecture, which is not involved in this scenario). Caches have the next lowest latency after the registers, since they are also on the R10000 chip (primary cache) or tightly-coupled to its processor on a daughterboard (secondary cache).

Each CPU has a primary instruction cache, a primary data cache, and a secondary cache which is used to hold both instructions and data.

Memory

Memory can be either local or remote. The access is **local** if the address of the memory reference is to an address in that piece of memory space assigned to the node the processor is on. The access is **remote** if the address of the memory reference is to anywhere else in memory, all of which has been assigned to other nodes. In the diagram, local memory is the section of main memory assigned to Node 1, which means this area of memory is local to Processor 0 (and Processor

2-17

22jul1998

TR-IKI rev 0.7b SGI Proprietary

1).

Remote caches

Remote caches may be holding copies of a given memory block. If the requesting processor is writing, all other cache copies must be invalidated. None of this is a memory latency issue for the processor doing the writing.

If the requesting processor is reading, memory latency will be an issue only if some other processor has the most up-to-date copy of the requested location. If this is true, then that other processor's cached copy of the information must first be written to disk, before the requesting processor can access that information for reading. In the diagram, the blocks labeled "cache" on Nodes 2 and 3 are remote to Node 1 (as are all the rest of the caches on any module in the machine).

Caches are used to reduce the amount of time it takes to access memory (also known as a memory's latency) by moving faster memory physically close to, or even onto, the processor.

While data only exists in either local or remote memory, copies of the data can exist in various processor caches. Keeping these copies consistent is the responsibility of the logic of the various hubs. This logic is collectively referred to as a *cache-coherence protocol*.

2-17.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

More About Cache

Non-Blocking Cache

In a typical implementation, the processor executes out of the cache until a cache miss is taken. A number of cycles elapse before data is returned to the processor and placed in the on-chip cache, allowing execution to resume. This type of implementation is referred to as a blocking cache because the cache cannot be accessed again until the cache miss is resolved.

Non-blocking caches allow subsequent cache accesses to continue even though a cache miss has occurred. Locating cache misses as early as possible and performing the required steps to solve them is crucial in increasing overall cache system performance.

The major advantage of a non-blocking cache is the ability to stack memory references by queuing up multiple cache misses and servicing them simultaneously. The sooner the hardware can begin servicing the cache miss, the sooner data can be returned.

Cache Types

- **Primary Data Cache**
- **Primary Instruction Cache**
- **Secondary Cache (For Both Data and Instructions)**

The Primary Caches for data and instructions are a subset of the larger Secondary Cache which can contain both. All three caches use a least-recently-used (LRU) replacement algorithm.

Primary Data Cache

The primary data cache of the R10000 microprocessor is 32K bytes in size and is arranged as two identical 16K-byte banks. The cache is two-way interleaved, allowing memory accesses to be overlapped. Each of the two banks is two-way set associative (that is, two cache blocks are assigned to each set). Cache line size is 32 bytes. The data cache uses a fixed block size of 8 words.

The data cache uses a write back protocol, which means a cache store writes data into the cache instead of writing it directly to memory. Sometime later this data is independently written to memory.

Write back from the primary data cache goes to the secondary cache, and write back from the secondary cache goes to main memory, through the system interface. The primary data cache is written back to the secondary cache before the secondary cache is written back to the system interface.

The data cache is indexed with a virtual address and tagged with a physical address. Each primary cache block is in one of the following four states:

- Invalid
- CleanExclusive
- DirtyExclusive
- Shared

A primary data cache block is said to be Inconsistent when the data in the primary cache has been modified from the corresponding data in the secondary cache. The primary data cache is maintained as a subset of the secondary cache where the state of a block in the primary data cache always matches the state of the corresponding block in the secondary cache.

A data cache block can be changed from one state to another as a result of any one of the following events:

- primary data cache read/write miss
- primary data cache write hit
- subset enforcement
- a CACHE instruction
- external intervention shared request
- intervention exclusive request
- invalidate request

Primary Instruction Cache

The instruction cache is 32K Bytes and is two-way set associative. Instructions are partially decoded before being placed in the instruction cache. Four extra bits are appended to each instruction to identify which execution unit the instruction will be dispatched to. The instruction cache line size is 64 bytes. The instruction cache has a fixed block size of 16 words and is two-way set associative.

The instruction cache is indexed with a virtual address and tagged with a physical address.

Each instruction cache block is in one of the following two states:

- Invalid
- Valid

An instruction cache block can be changed from one state to the other as a result of any one of the following events:

- a primary instruction cache read miss
- subset property enforcement
- any of various CACHE instructions
- external intervention exclusive and invalidate requests

Secondary Cache (for Data and Instructions)

The R10000 processor must have an external secondary cache, ranging in size from 512 Kbytes to 16 Mbytes, in powers of 2, as set by the SCSIZE mode bit. The SCBLKSIZE mode bit selects a block size of either 16 or 32 words. Secondary cache line size is programmable at either 64 or 128 bytes.

The secondary cache interface of the R10000 microprocessor provides a 128-bit data bus which can operate at a maximum of 200 MHz, yielding a peak data transfer rate of 3.2 GBytes/second. The secondary cache is two-way set associative (that is, two cache blocks are assigned to each set).

Each secondary cache block is in one of the following four states:

- Invalid
- CleanExclusive
- DirtyExclusive
- Shared

A secondary cache block can be changed from one state to another as a result of any of the following events:

- primary cache read/write miss
- primary cache write hit to a Shared or CleanExclusive block
- secondary cache read miss
- secondary cache write hit to a Shared or CleanExclusive block
- a CACHE instruction
- external intervention shared request
- intervention exclusive request
- invalidate request

Determining What Hardware the System is Running

hinv - Hardware inventory command

The "hinv" command displays the contents of the system hardware inventory table. This table is created each time the system is booted and contains entries describing various pieces of hardware in the system. The items in the table include main memory size, cache sizes, floating point unit, and disk drives. Without arguments, the hinv command displays a one line description of each entry in the table.

Determining What Memory Looks Like

The following are useful for determining how memory is being used, how much is "userland", how much is "systemland", how much was the default allocation for cache, how much cache is there now, what's in it, etc. Do a "man" or see web pages for these:

- ps
- top
- gr_top
- osview
- /usr/sbin/osview
- gr_osview
- gmemusage (originally called "bloatview")

**Module 3: Memory and Addressing: Pages, TLB's, and
Addressing, From a Hardware Perspective**

Memory and Addressing from a Hardware Perspective

Since IRIX systems are *virtual memory* systems, that is, an entire process need *not* be completely in memory to run, there are special hardware and software considerations involved in translating and calculating a process's actual address in physical memory, and determining if the requested address is something which needs to be brought in from disk before the process can continue.

This section provides information about how the Cray Origin2000 hardware references memory and handles addressing.

By the end of this section, the student should be able to:

- Explain how virtual memory systems differ from physical memory systems
- Explain the concept of memory divided into pages
- Describe the function of the TLB
- Describe the sequence of events involved in satisfying a memory request
- Describe the four important memory segment types for 64-bit architectures
- Interpret the segment type from a virtual address

HARDWARE MEMORY

Pages, and TLB's

Introductory Concepts About Pages

- **Memory is managed in *pages***

With IRIX, memory is managed in amounts called "pages". Pages are typically 16Kbytes in size, although the size can vary. To a processor, main memory appears as a single addressable space containing many pages.

- **IRIX is a *virtual memory* operating system**

While there are only so many actual physical pages of memory on a machine, the IRIX operating system uses a methodology of "virtual memory", which allows the memory requirements of all the processes on the machine to add up to more actual pages than the physical machine contains.

- **A process does not need all of its pages in physical memory**

Each process is assigned to a range of *virtual* addresses, some of which are mapped to physical memory pages with actual data when the process is first created. The system requires only those pages a process is actually referencing to be physically present in memory, while unreferenced pages can remain as "virtual" addresses, that is, no physical page of memory has been allocated for this page, or contains this page's data.

- **Process pages do not need to be contiguous in *physical* memory**

If the process never references a virtual page, then a physical page will never be assigned for it. If and when a process needs to reference a "virtual" page, then that page will be mapped to a physical page and the data will be brought into main memory. Although the physical pages of a process do not need to be contiguous in *physical* memory, the operating system will organize the *virtual process addresses* into a contiguous *virtual process image*.

Introductory Concepts About the TLB

- **"TLB" - Translation Lookaside Buffer**

The Cray Origin2000 R10000 MIPS processor chip hardware contains an array of 128 Translation Lookaside Buffer (TLB) entries. The R4000 and R5000 chips hold 64 TLB register entries.

- **The TLB translates virtual addresses to physical addresses**

The function of the TLB is to translate virtual addresses to physical addresses.

The TLB is a virtual cache. The "data" cached by each TLB entry is the physical page number and page access permissions that matches a particular virtual page address.

- **"TLB Hit" - the physical page reference is in the TLB already**

When a processor wants to reference one of a process's virtual addresses, it looks first in its TLB to calculate what physical page matches the virtual address reference. If the process has already referenced this virtual page, and the matching physical page reference is still loaded in the TLB, then the physical offset into this page address can be calculated immediately, and the data can be accessed quickly, and passed along to the CPU's secondary and primary caches.

- "TLB Miss" - virtual page reference does not match a TLB entry

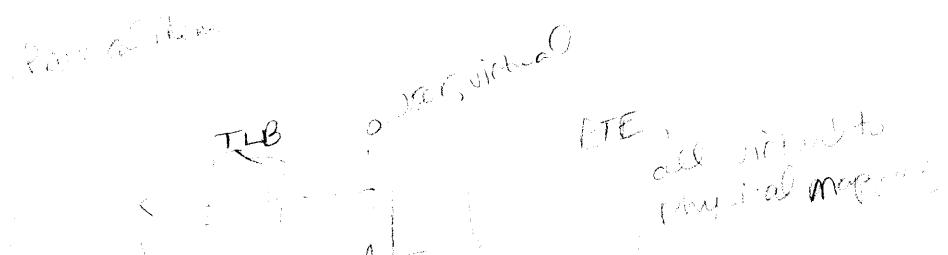
When a processor wants to reference a process's virtual address and does not find a matching TLB entry, the CPU must do a context switch to the operating system kernel code. The kernel will check to see if there is an existing physical page loaded with process data which matches the virtual address.

- If the physical page is currently loaded in memory...

The kernel will do the calculations to translate the virtual process address to a physical memory page. If there is a valid translation, then the kernel will load a TLB entry to describe that page, and restart the instruction. This time, when the CPU tries to match the process's virtual address request with a TLB entry, the process gets a "TLB hit", and is able to reference the requested address.

- If the physical page is *NOT* currently loaded in memory... (kernel uses PTE)

In this case the kernel determines that there is no physical page loaded with process data, that matches the virtual address the process now wants to reference. This is called a *page fault*. At this point, the kernel must calculate the disk location of the page containing the address the process wants to reference, and then move that page of disk information into a physical memory page. Once this is accomplished, the kernel can calculate a valid physical-to-virtual address translation, and load the TLB with a description of that page. When the process instruction is restarted, the process (finally) gets a "TLB hit", and is able to reference the requested address.



Memory Management Philosophies

Real Memory Machines and Swapping

One of the major concerns for the operating system is how it manages the finite amount of physical memory installed in the system hardware. The aggregate amount of memory needed by all active processes on the system is constantly changing and generally is far greater than available physical memory.

On "real" memory machines, a process must be entirely located in physical machine memory, in order to run.

Earlier versions of UNIX used a method called *swapping* to manage main memory. With this method, whole processes were swapped from memory to disk to make room for other processes that needed to run. Swapping was done by a special process called the *swapper* or *sched* (short for *scheduler*), which always had a PID=0.

"Swapper" is still the first process created on most UNIX-based systems. When the system first comes up, the first process, PID(0) does a lot of system initialization. When process 0 is finished, it renames itself "*sched*" and jumps into a loop of code which is the process swapping routine. This routine sleeps, and wakes up when there is work to do.

In 6.5, swapper is now a service thread

Virtual Memory Machines and Paging

UNIX System V Release 4 adopted a concept referred to as *virtual memory*. A virtual machine allows programmers to ignore the physical layout and size of machine memory. A program is written to reference virtual addresses for both instructions and data, thus relieving the programmer from concern as to where things are physically located in memory.

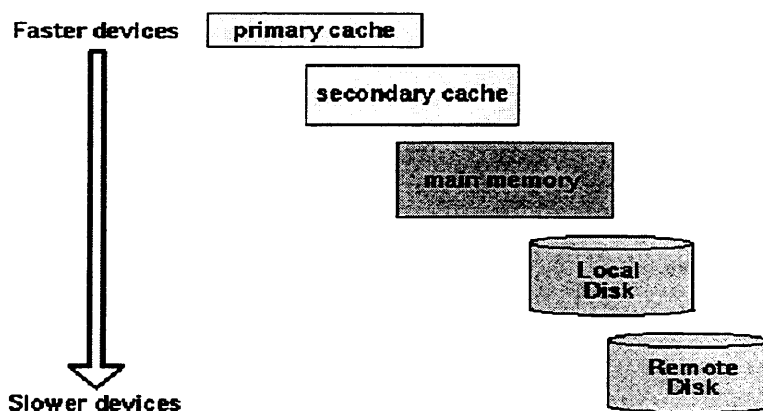
On IRIX, an entire process does *not* have to be completely in memory to run. Instead, only those pieces, or "pages", of the process needed to execute are required to be physically in machine memory.

Virtual memory systems:

- Give the illusion that there is more memory available than physically installed on machine.
- Can run programs that are larger than physical memory.
- Do not require the process to be entirely in physical memory to run.
- Require a translation mechanism to convert virtual memory addresses to physical addresses at run time.

Where Are the Addresses the Process Isn't Using?

Virtual memory is implemented using a hierarchical-storage scheme, as shown below. The parts of a process which are not being used and which will not fit in memory are held on secondary storage devices such as disk or remote disks accessible over the network.



The subsystems of the kernel and the hardware that cooperate to translate virtual to physical addresses comprise the *memory management subsystem*.

This section focuses on the hardware aspects.

Memory pages

IRIX implements a memory management architecture based upon *pages*. The kernel divides all of physical memory into a set of equal-sized blocks called *pages*. The size of each page is defined by the hardware. For IRIX-based systems a page is a multiple of 4 KB, and typically is 16 KB in size.

The `sysconf(1)` command, using an argument of either `PAGESIZE` or `PAGE_SIZE`, can be used to display the definition of page size on an IRIX system.

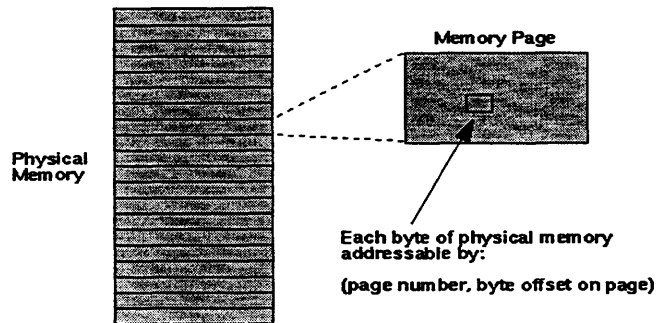
Below is an example of a 64 GB Origin2000 system and a calculation showing the number of memory pages defined on the system.

$$\begin{aligned}
 1 \text{ KB} &= 1,024 \text{ bytes} = 2^{10} \text{ bytes} \\
 1 \text{ MB} &= 1,048,576 \text{ bytes} = 2^{20} \text{ bytes} \\
 1 \text{ GB} &= 2^{30} \text{ bytes} \\
 \text{Origin2000 (example)} \\
 \text{memory size} &= 64 \text{ GB} \\
 \# \text{ pages} &= \frac{\text{memory size}}{\text{size/page}} = \frac{64 * 2^{30}}{16 * 2^{10}} = 4 * 2^{20} = 4,194,304 \text{ pages}
 \end{aligned}$$

HARDWARE ADDRESSING

All Addresses = (Page Number + Byte Offset)

Every addressable location in physical memory is contained in a memory page. Therefore, every memory location (byte) can be addressed by a pair of values: (page number, byte offset on page)



PFN
 page frame number -
 Physical page #
 comp of
 (node, bank, + page)

Cray Origin2000 Memory Hierarchy and Latency

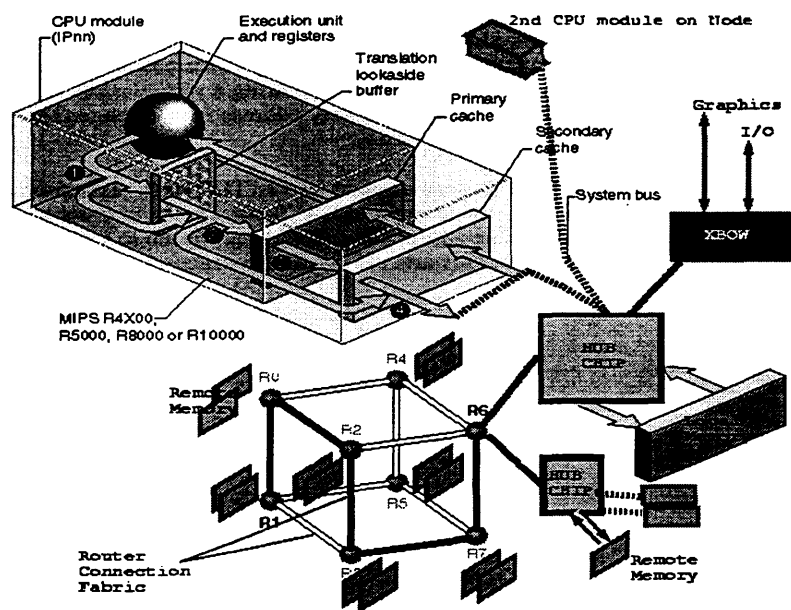
Page migration hardware moves data into memory closer to a processor that frequently uses it, in order to reduce *memory latency*, that is, how long it takes for a processor to access memory contents.

Remember from the Cray Origin2000 Architecture lesson which contained the memory latency hierarchy explanation and diagram, that memory contents are accessed, in order of increasing memory latency (increasingly slow access), as follows:

- Processor registers for this CPU
- Primary Cache for this CPU
- Secondary Cache for this CPU
- Local Memory (the memory assigned to this node)
- Remote Memory (the rest of the system's memory)
- Remote Caches of other CPU's (the condition of the data in another CPU's cache could require time-consuming operations, such as a write, before this CPU can access the data)

Much of the rest of this section overviews how an address is found.

Address Request Sequence



A CPU examines an instruction, and isolates that part of it which represents the address of the page, and the offset into that page, of the data that the CPU needs. This address might be something like the address of an instruction to fetch, or the address of an operand of an instruction. Then the CPU goes through the following steps in order to find that address.

1. The virtual address of the needed data is formed in the processor execution or instruction-fetch unit. Most addresses are then mapped from virtual to real through the Translation Lookaside Buffer (TLB). This process may have had a "TLB miss" if the virtual-to-physical mapping was not already in the TLB. At that point, the CPU had to exchange into kernel context in order to determine the physical address and then load it into the TLB. One way or another, at this point the TLB has a virtual-to-physical address mapping of the address the process wants, and the CPU 'knows' what physical page of memory it must access.
2. Most addresses are presented to the primary instruction or primary data caches, depending on what is being addressed. These caches are in the processor chip. If a copy of the data with that address is found, it is returned immediately.
3. When the primary cache does not contain the data, the address is presented to the secondary cache, which is used to hold both data and instructions. If the secondary cache contains a copy of the data, the data is returned immediately.
4. When the secondary cache does not contain the data, the physical address reference is placed on the system bus and handed over to the HUB chip. The HUB knows which areas of memory have been assigned to which nodes, which area of memory has been assigned as "local" to this node, and which nodes are attached to which router connections. The HUB acts as a switch, and directs the request either to this node chip's local memory, or whatever remote memory address is appropriate.
5. When the HUB chip recognizes that local memory does not contain the data, the address passes out through the "connection fabric", that is, through router connections to other nodes on this, or other hypercubes in the system, to a memory module in another node, from which the data is returned.

TLB Misses

Each TLB entry on an R10000 chip describes two pages whose *virtual* addresses are adjacent, and maps each to the actual physical page of real memory containing the data. Remember that, although the virtual addresses are contiguous, the two corresponding physical pages probably are not. On an R4000 or R5000 chip, a TLB entry describes only a single page of memory. When a CPU tries to execute something relating to a process's address, it is using a *virtual* address. If the address falls in a page described by a TLB entry, the TLB supplies the physical memory address for that page. The translated address, now physical instead of virtual, is passed on to the secondary cache.

When the input address is not covered by any active TLB entry, the MIPS processor generates a "TLB miss" exception, which means that the CPU stops executing user code, and changes its context to execute IRIX kernel code, in order to handle this TLB-related situation. The kernel inspects the address. When the address has a valid translation to some page in the address space, the kernel loads a TLB entry to describe that page, and restarts the original instruction.

Two Types of "TLB Miss"

There are two kinds of "TLB miss" situations.

In one case, the CPU examines the TLB, and does not find a physical page reference because the page has never been loaded into memory to begin with.

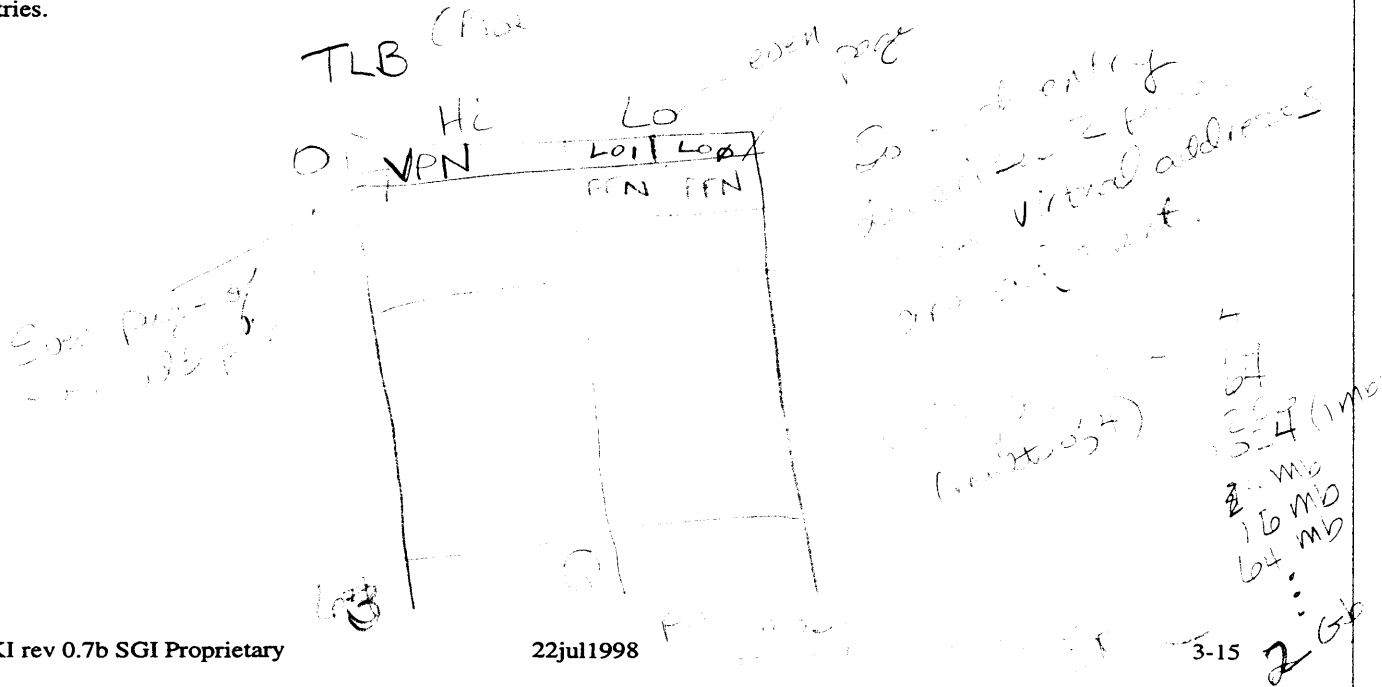
In the second kind of TLB miss, the page *is* in physical memory, but isn't in the TLB for some reason, for example, that reference may have been loaded in the TLB earlier, but eventually stopped being referenced, aged, and was overwritten.

The TLB is hardware. Handling a TLB miss is solved with software. There is more detail on how the kernel handles each of these two TLB miss situations in a later section.

TLB Size

The size of the TLB is important for performance. As long as the CPU finds virtual-to-physical address mappings readily convenient in the TLB, the process can continue to execute (until some other even forces a context switch).

The TLB associated with the R10000 chip holds 128 entries. The TLB associated with the R4000 and R5000 chips hold 64 entries.



Coprocessor 0 and the TLB

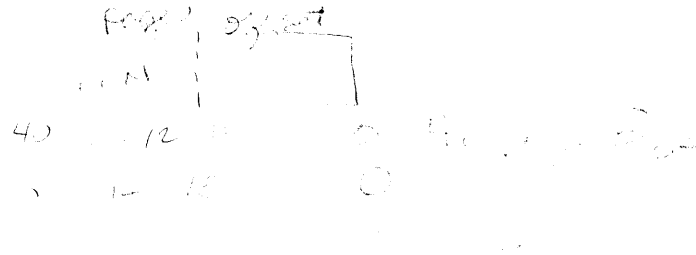
Coprocessors are alternate execution units, with register files separate from the CPU.

The MIPS architecture provides an abstraction for up to 4 coprocessor units, numbered 0 to 3. Each architecture level defines some of these coprocessors. Coprocessor 1 is used for the floating-point unit. Coprocessor 0 is always used for system control. Other coprocessors are architecturally valid, but do not have a reserved use. Some coprocessors are not defined and their opcodes are either reserved or used for other purposes.

Many of the coprocessor 0 registers are related to the TLB and exception processing, as shown below.

Register No.	Register Name	Description
0	Index	Programmable register to select TLB entry for reading or writing
1	Random	Pseudo-random counter for TLB replacement
2	EntryLo0	Low half of TLB entry for even VFN (Physical page number)
3	EntryLo1	Low half of TLB entry for odd VFN (Physical page number)
4	Context	Pointer to kernel virtual FTE table in 32-bit addressing mode
5	PageMask	Mask that sets the TLB page size
6	Wired	Number of wired TLB entries (lowest TLB entries not used for random replacement)
7	Undefined	Undefined
8	BadVAddr	Bad virtual address
9	Count	Timer count
10	EntryHi	High half of TLB entry (Virtual page number and ASID)
11	Compare	Timer compare
12	Status	Processor Status Register
13	Cause	Cause of the last exception taken
14	EPC	Exception Program Counter
15	FRid	Processor Revision Identifier
16	Config	Configuration Register (secondary cache size, etc.)
17	LLAddr	Load Linked memory address
18	WatchLo	Memory reference trap address (low bits Adr[39:32])
19	WatchHi	Memory reference trap address (high bits Adr[31:3])
20	XContext	Pointer to kernel virtual FTE table in 64-bit addressing mode
21	FrameMask	Mask the physical addresses of entries which are written into the TLB
22	BrDiag	Branch Diagnostic register
23	Undefined	Undefined
24	Undefined	Undefined
25	FC	Performance Counters
26	ECC	Secondary cache ECC and primary cache parity
27	CacheErr	Cache Error and Status register
28	TagLo	Cache Tag register - low bits
29	TagHi	Cache Tag register - high bits
30	ErrorEPC	Error Exception Program Counter

Page Mask ²³
 000000
 000011
 000111



page mask indicates difference
 of bits in each bit position from page to
 next page

Binary, Hexadecimal, and Decimal Address Conversions

Below is a brief reminder of the bit pattern significance for hexadecimal addressing.

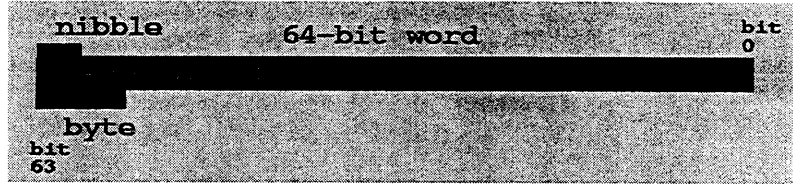
Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

The 64-Bit Address Space and "Segments"

The 64-bit mode is an upward extension of 32-bit mode. All MIPS processors from the R4000 on support 64-bit mode.

There are four bits to one nibble, and two nibbles to one (eight-bit) byte.

There are eight bytes to one (sixty-four bit) word.



The MIPS hardware divides the address space of system memory into *segments* based on the most significant bits, and treats each segment differently. The ranges are shown graphically, below.

These major segments define only a fraction of the 64-bit space. Most of the possible addresses are undefined and cause an addressing exception (segmentation fault) if used.

When operating in 64-bit mode, the MIPS architecture uses addresses that are 64-bit unsigned integers from (hexadecimal):

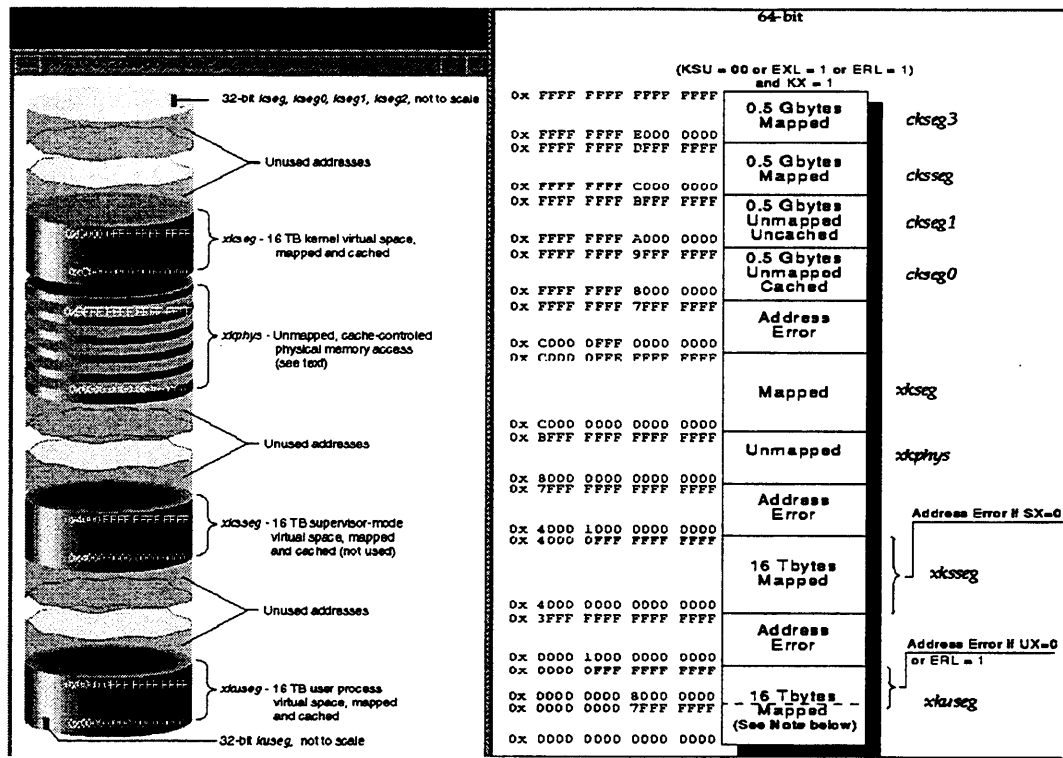
0x0000 0000 0000 0000 to 0xFFFF FFFF FFFF FFFF.

This is an immense span of numbers - if it were drawn to a scale of 1 millimeter per terabyte, the drawing would be 16.8 kilometers long (just over 10 miles).

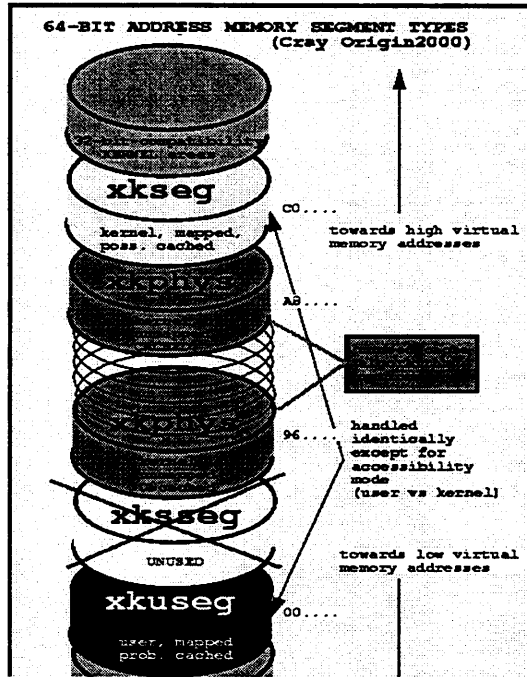
Illustrations of Segment Types

Both of the next two illustrations show memory divided into the various segment types. The illustration on the left shows a better representation of the segment types. The illustration on the right gives a better representation of which hexadecimal addresses map to those segment types.

These illustrations are a good starting point for understanding the different segment types, but some of what is shown is somewhat confusing. See the explanation of segment types and characteristics, which follows.



Below is a simplified version of the earlier pair of illustrations, which summarizes the possible memory segment types for a Cray Origin2000 architecture.



3-22

22jul1998

TR-IKI rev 0.7b SGI Proprietary



3-22.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Segment Characteristics

There are a number of different segment types shown in the previous illustrations. These segments differ, depending on two major characteristics:

- whether or not the address must be translated, or "mapped", from a virtual reference to a physical memory reference by the translation lookaside buffer (TLB).
- whether an address can be accessed when the CPU is operating in user mode or in kernel mode.

And there is an additional difference which is not segment-specific, but address-specific:

- whether this particular address will be cached or not.

For all segment types, each address is potentially cacheable, so whether this is something to be considered about this particular address (or not) must be checked. This last difference does not distinguish segments types, it is a distinction about the handling of each specific address, regardless of segment type. Whether to cache an address or not, is determined by bit settings (explained below).

Segment Types Overview

The simplified diagram of memory segment areas attempts to display these essential concepts:

- 64-bit machines are compatible with 32-bit machines
 - (probably) ignore these segments
- Memory access is divided into kernel and user areas, based on CPU mode
 - Type is determined by the high-order bit (bit 63)
 - 0 = user
 - 1 = kernel
 - Four possible areas (segment types)
 - Ignore supervisory mode related references [xksseg]
- User area:
 - **xkuseg - virtual user memory**
 - Virtual to physical address translation done through TLB ("mapped")
 - High order bits 63:56 = "00"
 - Might be cached
- Kernel areas :
 - **xksegs - virtual kernel memory**
 - Virtual to physical address translation done through TLB ("mapped")
 - High order bits 63:56 = "C0"
 - Might be cached
 - **xkphys - physical kernel memory**
 - Low-order 44 bits used as direct physical address ("unmapped") (no TLB)
 - Six subdivisions based on caching algorithms
 - only two used, ignore the rest
 - If high order byte (bits 63:56) = "A8", xkphys address might be cached
 - If high order byte (bits 63:56) = "96", xkphys address will never be cached

Table of Cray Origin2000 Segment Types and Characteristics

The table below is a summation of the bit patterns, and mapping methodology, and caching characteristics, of the four memory segment areas of most interest on a 64-bit machine.

The "6" in the uncached xkphys segment high bits "96" is the only part of this segment addressing scheme that is Origin2000 specific. The interpretation of the bits, and the implementation of these characteristics, is heavily influenced by the CPU and also a little by the general SNO architecture.

Table 3-0: Segment Types and Characteristics for Cray Origin2000 Architecture

Segment Address Space Type	High Order Bit (BIT 63)	Second Highest Bit (BIT 62)	High Two Nibbles: (BITS 63:56) HEX : BINARY:	Mapped Address (Through TLB)	User Accessible	Kernel Accessible	Cache Algorithm Determined By:
xkseg (k2seg) kernel, mapped, poss. cached	<i>1</i> (kernel)	<i>1</i> (mapped)	C0 <i>1100 0000</i>	YES	NO	YES	TLB
xkphys (k0seg) kernel, unmapped, CACHED	<i>1</i> (kernel)	<i>0</i> (unmapped)	A8 <i>1010 1000</i> 61:59	NO	NO	YES	BITS 61:59 (bits indicate this address will be cached)
xkphys (k1seg) kernel, unmapped, UNCACHED	<i>1</i> (kernel)	<i>0</i> (unmapped)	96 <i>1001 0110</i> 61:59	NO	NO	YES	BITS 61:59 (bits indicate this address will not be cached)
xksseg		U	N	U	S	E	D
xkuseg user, mapped, prob. cached	<i>0</i> (user)	<i>0</i> (mapped)	00 <i>0000 0000</i>	YES	YES	YES	TLB

32-Bit Compatibility Areas

You can probably ignore all the 32-bit compatibility addresses on your machine, unless there is a user code running in that context.

On a 64-bit machine, the beginning and end of memory have areas to make the machine compatible with 32-bit architectures. These areas are listed in the illustrations above as [kseg, kseg0, kseg1, kseg2, and kseg3] in one drawing, and [cksseg, ckseg0, ckseg1, and ckseg3] in the other. Comparing the two illustrations may lead to some confusion.

In the right-hand illustration, you may notice that while there is a kxsseg, ckseg0, ckseg1, and ckseg3 area, there is no "ckseg2" area. This is because the ckseg2 area was split into the ckseg3 and kxsseg areas.

In the left-hand illustration, you may notice that in this case the areas are numbered sequentially, kseg, kseg0, kseg1, kseg2, and even kseg3, but there is no "kxsseg" area, although you may occasionally find it used to refer to "kernel mapped space" (more on "mapped" versus "unmapped" below).

Addresses Accessed Based on CPU Mode

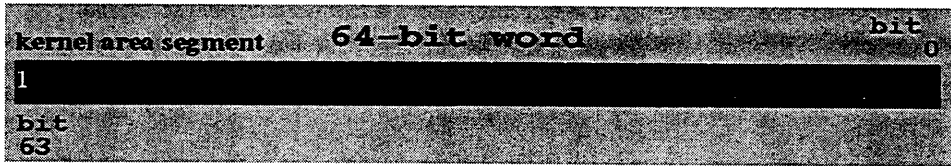
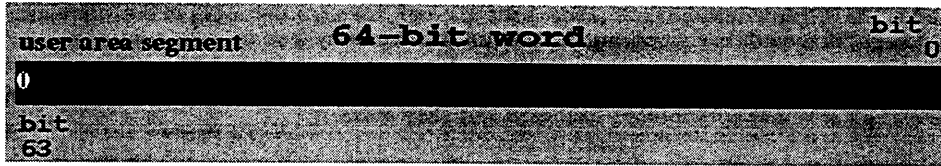
The 64-bit compatible memory addresses are divided into kernel areas, which only CPU's running in *kernel mode* can access, and the user area, which CPU's running in either *kernel* or *user mode* can access.

You will probably encounter references to a third mode, *supervisory mode*, as well. This is a privilege level somewhere between user mode and kernel mode. This mode is NOT implemented. You can ignore all references to it in both documentation and in the code (it was easier to leave the code in, than to remove it). You can ignore the area of memory addresses, **xksseg**, devoted to it.

There are really only four different types of memory segments you will probably need to know about. These are the three kernel-only address areas composed of **xkseg**, and two subdivisions of the **xkphys** area, and the fourth segment type composed of the user-specific area **xkuseg**.

Cray Origin2000 Segment Types

When a processor references an address, it looks at the high bits of the address to determine whether the address falls into user-accessible memory addresses (the high-order bit, bit 63, is a "0"), or kernel-only memory addresses (the high-order bit, bit 63, is a "1").



Most addresses presented to the CPU are virtual addresses and must be "mapped", or translated, through the TLB, into physical memory references.

Some addresses presented to the CPU are used as "direct", or "unmapped", references to a physical location, that is, the address is not translated through the TLB, but is interpreted instead as a reference to a physical area of memory.

Interpreting the Segment Type From the Virtual Address

Each 64-bit address value is treated as shown below



The two most significant bits select the major segment.

The `xkuseg`, `xkphys`, and `xksegs` segment types are discussed below. The `xksseg` segment is not utilized, so it is ignored.

The size of a page of virtual memory can vary from system to system and release to release, so always determine it dynamically. In a user-level program, call the `getpagesize()` function (see the `getpagesize(2)` reference page). In a kernel-level driver, use the `ptob()` kernel function (see the `ptob(D3)` reference page) or the constant `NBPP` (Number of Bytes Per virtual Page) declared in `/usr/include/sys/immu.h`.

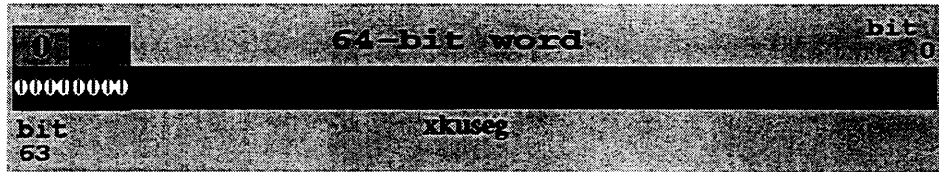
When the page size is 16 KB, bits 13:0 of the address represent the offset within the page, and bits 39:14 select a Virtual Page Number (VPN) from the 226, or 64 M, pages in the virtual segment..

User Address Area Segment

`xkuseg` - Virtual User Memory - mapped, probably cached

- **Distinguishing Bit Pattern?**

If the high-order bit of an address, bit 63, is a 0, then the address refers to the user memory segment.



In fact, the upper two "nibbles" (ie, the upper 8 bits, 4 bits per nibble, same bits as the upper byte) are always all zeros for user address references. (It may actually be the case that bits 62:56 could be something other than 0. It seems to be the case that this is such a large virtual address value, this has never been tested.)

- **Accessible to Which CPU Modes?**

These addresses are the only ones CPU's in user mode can access. CPU's in kernel mode can access these areas, as well as the kernel-only memory addresses. The xkuseg area, and the xkseg area (kernel mapped, possibly cached, see below), are treated identically, except that only the kernel can access the xkseg area.

- **What's it used for?**

The xkuseg area is the area devoted to user process space. User address space takes up roughly half of memory (about 16 terabytes).

- **Mapped or Unmapped?**

All user area addresses are considered *mapped* addresses. This means that the 64-bit address the CPU is examining is a

virtual address and cannot be used "as-is" to find an actual physical location. The CPU must go through the TLB in order to translate the address into a real physical memory reference.

The kernel creates a unique address space for each user process. Of the 226 possible pages in a process's address space, most are typically unassigned, and many are shared pages of program text from dynamic shared objects (DSOs) that are mapped into the address space of every process that needs them.

The Origin2000 architecture adds the complication that the location of a page, relative to the location where the process executes, has an effect on the performance of the process. The kernel uses a variety of strategies to locate pages of memory in the same node as the CPU that is running the process.

- **Cached or Uncached?**

User area addresses references are probably going to be cached. This means that CPUs must be concerned with cache coherency issues before loading or storing user area addresses.

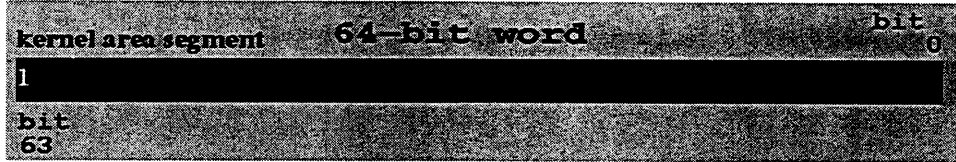
Any attempt to read that memory location must confirm that there is not a version which was read into cache and changed, somewhere else in the machine. Such an occurrence would make the memory version incorrect, since the memory version would not be the most recent version of the address's contents.

Any attempt to write to that memory location will make other cached versions on the machine outdated and invalid.

It's possible that a user could access the xkuseg area with uncached reference, but the user would have to do special `syssgi` calls that are only used by SGI diagnostics to get uncached access to memory. Uncached access to the xkuseg segment is Of allowed by the architecture and CPU, but the kernel chooses not to use the hardware in this way.

Kernel Address Area Segments

For all kernel-only address area segments, the high-order bit (bit 63) is a "1".



There are two major areas of kernel memory, **xkseg** and **xkphys**. The kernel distinguishes which of the two segment types an address falls into, by examining additional bits in the address.

The second-highest bit, bit 62, determines whether this address reference is to an **xkseg** area (bit 62 is a "1"), or an **xkphys** area (bit 62 is a "0").

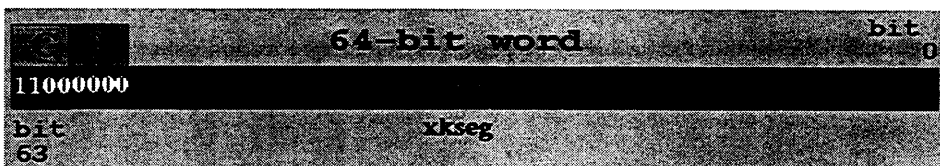
The **xkphys** area is further subdivided, based on cache-related issues. Bits 59, 60, and 61 (usually written "61:59") are examined to determine which caching algorithm to apply to memory in an **xkphys** segment.

There are only three kernel-specific segment types you probably need to know about, **xkseg**, and two of the subdivisions of **xkphys**.

xkseg - Virtual Kernel Memory - mapped, possibly cached

- Distinguishing Bit Pattern?

When bits 63:62 are "11", then the memory accessed is kernel virtual memory.



Only code that is part of the kernel can access this space, which is a 2 Terabyte segment starting at 0xC000 0000 0000. All addressing in the **xkseg** area starts with "C0" in the highest two nibbles (highest byte).

- Accessible to Which CPU Modes?

Only CPU's running in kernel context can access memory in the **xkseg** segment addresses.

- What's it used for?

This is the space in which the IRIX kernel allocates such objects as kernel stacks, per-process data that must be accessible on context switches, and user page tables. Certain important data structures may be replicated into each node for faster access.

This segment area is also the space in which kernel-level device drivers allocate memory, including automatic variables declared by loadable device drivers. The stack and data areas used by device drivers are in **xkseg**.

Since kernel space is mapped, addresses in the xkseg segment that are apparently contiguous need not be contiguous in physical memory. However, a device driver can allocate space that is both logically and physically contiguous, when that is required.

A driver has the ability to request memory allocation in a particular node, in order to make sure that data about a device is stored in the same node where the device is attached and where device interrupts are taken.

- **Mapped or Unmapped?**

References to this space are mapped (that is, translated through the TLB) and cached. The kernel uses the TLB to map kernel pages in memory as required, possibly in noncontiguous physical locations. The kernel passes the address through the TLB, and the TLB examines the virtual address, which it translates to a physical address.

This segment area is treated exactly the same way as the mapped user area (xkuseg) in terms of how the TLB translates virtual to physical address references. The difference is that, although pages in kernel space are mapped, they are always associated with real memory. Kernel pages are never paged to secondary storage.

- **Cached or Uncached?**

The TLB itself has some bits set which define how it will handle cache coherency issues, that is, the appropriate caching algorithm to use is determined by the TLB mapping. There are only two caching algorithm choices which are used. One is "don't cache it", and the other is "cacheable coherent update on write".

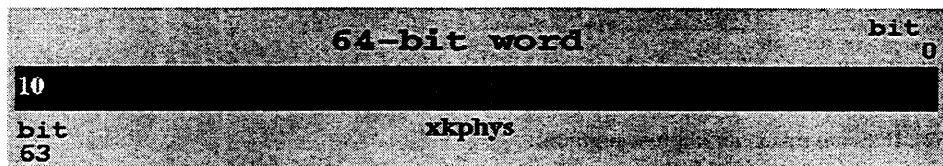
xkphys - Physical Kernel Memory

xkphys - unmapped, possibly CACHED

xkphys - unmapped, UNcached

- **Distinguishing Bit Pattern?**

When bits 63:62 are "10", then the memory accessed is kernel physical memory allocated in the xkphys segment.



For this particular segment type, three additional bits, bits 61, 60, and 59, are examined to determine whether cache residency is a relevant concern for memory addresses in this segment. There are six possible subdivisions of the xkphys memory area, based on what cache coherency algorithm to use for addresses in these sub-ranges, but only two of the subdivisions (and their caching algorithms) are actually used. See the "Cached or Uncached?" section, below.

- **Accessible to Which CPU Modes?**

Only CPU's running in kernel context can access memory in the xkseg segment addresses.

- **What's it used for?**

CPU's reference xkphys addresses in order to access kernel structures and data that will be needed "briefly", such as proc structures, vnode structures, buf structures, and all kernel dynamic data, all of which is managed by pfdats.

One-quarter of the 64-bit address space, that is, all addresses with bits 63:62 containing a bit pattern of "10", are devoted to special access to one or more 1 TB of physical address spaces.

- **Mapped or Unmapped?**

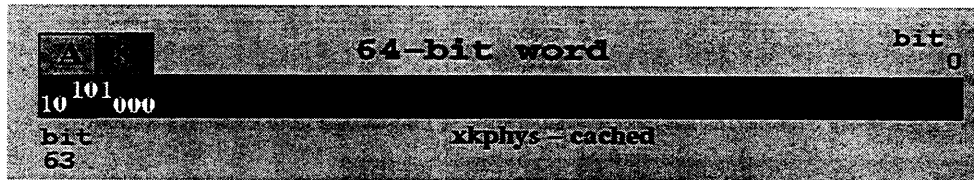
Direct references to this space are "unmapped", that is, the TLB is not involved in calculating the appropriate physical memory address location.

The entire 64 bits are a virtual address that is not really a physical memory address reference, but the processor knows how to decode it into a physical address. The physical address selected is taken directly from the lower 40 bits, bits 39:0, of that part of the word normally interpreted as a virtual page number and offset into the page. The three high order address bits discussed above, bits 61:59, determine if this memory reference is cached. Those 3 bits are part of the actual virtual address, which happens to map pretty straightforwardly to a physical address.

Access to addresses whose bits 56:40 are not equal to 0 cause an Address Error exception.

- **Cached or Uncached?**

- **xkphys - unmapped, possibly CACHED**



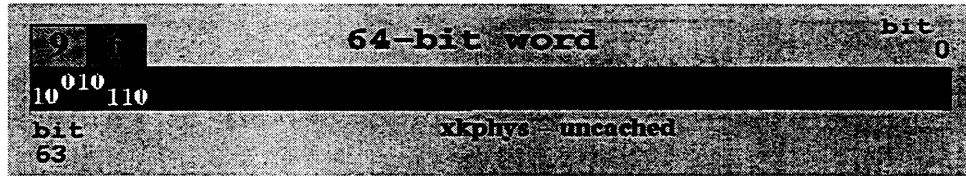
Addressing in the xkphys area that starts with "A8" in the highest two nibbles (highest byte) has a bit pattern of "1010 1000".

The first two bits, "10" (bits 63:62), indicate this is an xkphys segment.

The next three bits, "10 1" (bits 61:59), indicate which of the six possible xkphys caching algorithms is to be used, which, in this case, is the "cacheable coherent update on write" algorithm. Again, only two of the six possible algorithms are used.

This particular cached area of the xkphys segment starts at address 0xA800 0000 0000 0000, and goes through 0xAFFF FFFF FFFF FFFF, but it is highly probable you will only see addresses in this area that start with "A8".

● **xkphys - unmapped, UNcached**

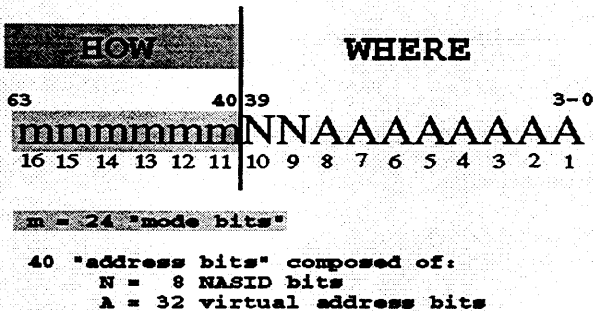


Addressing in the xkphys area that starts with "96" in the highest two nibbles (highest byte) has a bit pattern of "1001 0110".

The first two bits, "10" (bits 63:62), indicate this is an xkphys segment.

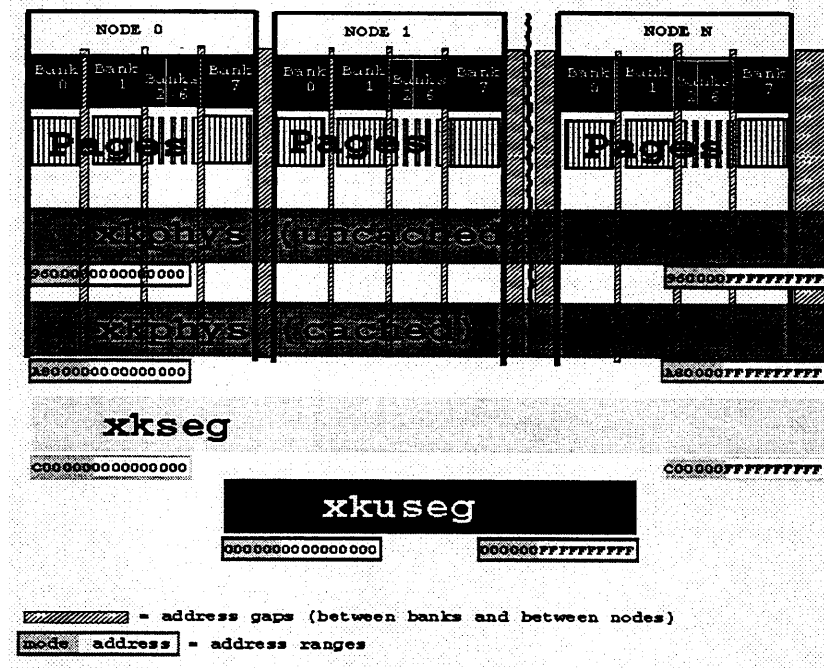
The next three bits, "01 0" (bits 61:59), indicate which of the six possible xkphys caching algorithms is to be used, which, in this case, is the bit pattern to specify that addressing in this particular range of xkphys should be "uncached". Again, only two of the six possible algorithms are used.

The 64-bit Word and the Virtual Address



1. Of the 64 bits in a virtual address, only 40 bits are actually address values.
2. The high order 24 bits can be considered "mode bits", which contain information about how to interpret the low order bits 39:0.
3. The NASID (Node Address Space ID) is the "power of two" at which each node's memory begins

A Different View of Memory Segments - Diagram



3-37

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Memory Segment Overview

- "Stacked tuna cans" view of virtual memory is misleading
 - the consecutive numbering across memory segment address ranges is misleading (00...00 -> 96...00 -> A8...00 -> C0...00)
 - the uncached xkphys (96...), cached xkphys (A8...), and xkseg (C0...) memory segments all describe the same range of physical memory pages
 - the xkuseg virtual memory addresses can refer to almost any pages of physical memory
 - the high order "mode bits" of a virtual address word indicate how a physical memory page will be referenced
- Physical memory has address gaps between banks and between nodes
 - these physical addresses do not exist
 - attempts to reference these non-existent addresses will cause errors
- There are 4 virtual memory segment types of primary interest
 - xkphy (cached) (A8..)
 - xkphys (uncached) (96...)
 - xkseg (C0...)
 - xkuseg (00...)

3-38

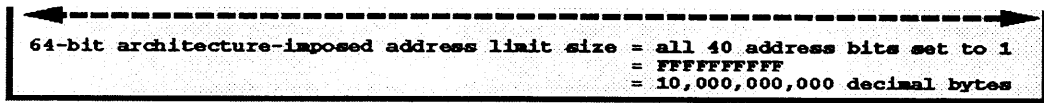
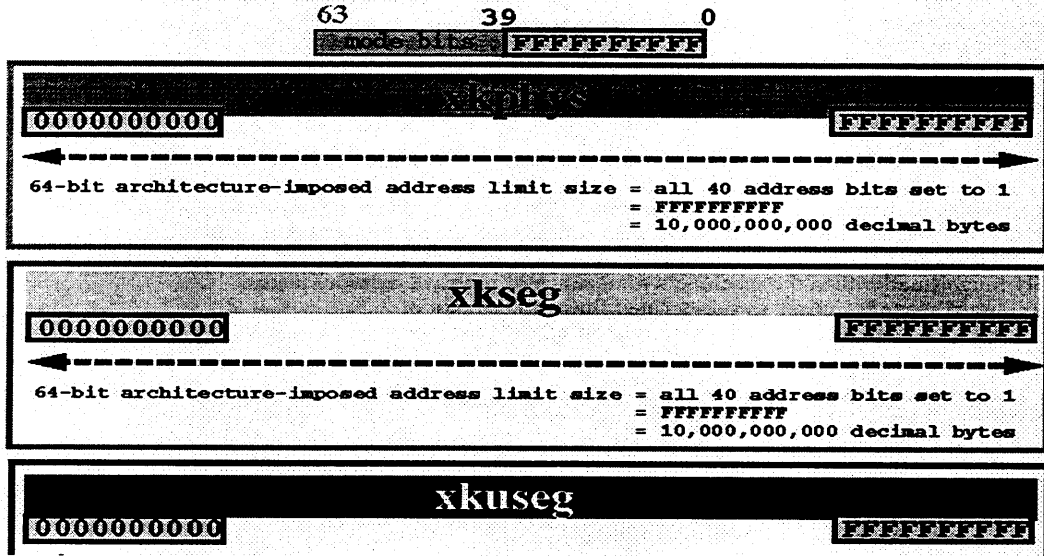
22jul1998

TR-IKI rev 0.7b SGI Proprietary

The following illustrations reference a 64-bit architecture.

For all types of memory segment addresses, there are three ranges to be considered:

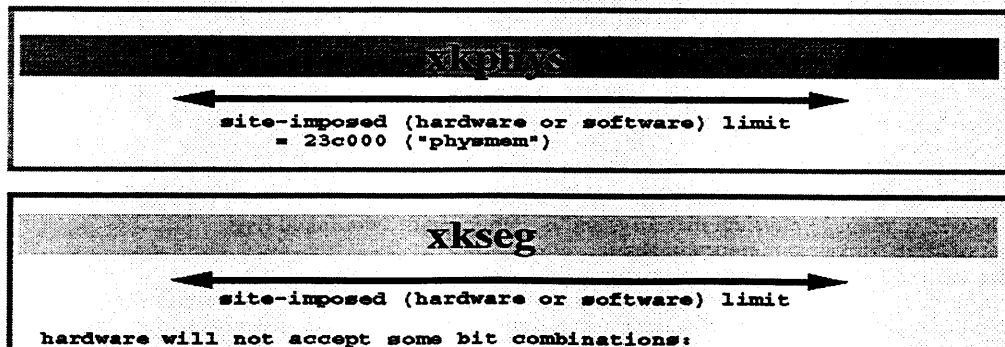
1) Maximum Address the Chip Size Allows



1) The maximum possible (physical or virtual) address size the chip architecture will allow (that is, if all of the bits the chip uses to reference an address are set to "1")

- of the 64 possible bits, bits 39:0 are used to specify an address
- 40 decimal bits constitute 10 hex characters of address
- An address 10 hex characters long could range from 0000000000 to FFFFFFFF

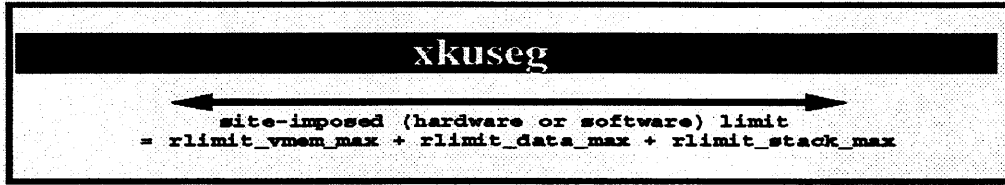
2) Highest Address Permitted by Configuration



```

largest hardware-legal virtual address = 7FFFFFFF ("K2SIZE")
software configuration tuning parameter limit:
largest software-legal segment size = 11E000 hex pages ("SYSSEGSZ")

```



2) The highest (physical or virtual) address size that has been configured as a software or hardware limit

- the actual number of banks of memory a site has purchased will limit the maximum *legitimate* physical address (xkphys), which is a number much smaller than the largest address the chip bit range could actually specify.
- the limit the site configures for total kernel space (xkseg) will be much smaller than what the chip bit range would allow, and probably much smaller than the machine's total range of physical memory pages.
- the user segment (xkuseg) space is limited to site-defined limits to the sum of $\text{rlimit_vmem_max} + \text{rlimit_data_max} + \text{rlimit_stack_max}$

3) Actual Number of Pages In Use



3-39.b

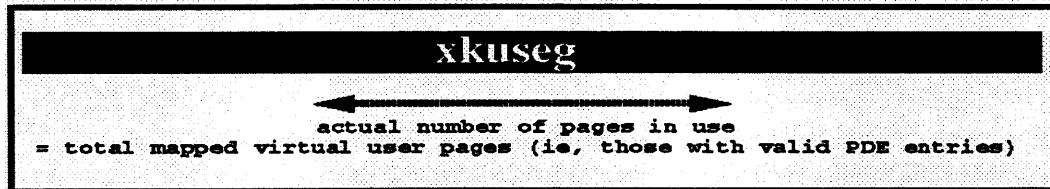
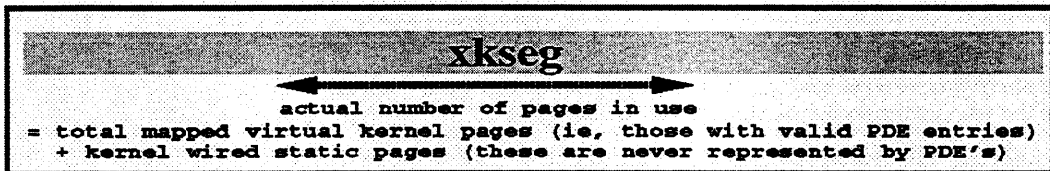
22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

actual number of pages in use
= (pfdat "in-use" pages) + (total pages used for the kernel static data)

```



3) The actual number of pages in use

- "xkphys" addresses
 - not all physical pages of memory will actually be in use at any given time
 - the "pfdat" table is used to manage physical memory, and is composed of two linked lists
 - the linked list of "free" physical memory pages
 - the linked list of "in-use" physical memory pages
 - *almost* all of the physical pages available on the system are listed in the pfdat tables. The physical pages used for the single 16Mbyte virtual page of kernel static information are *not* referenced with pfdat structures.

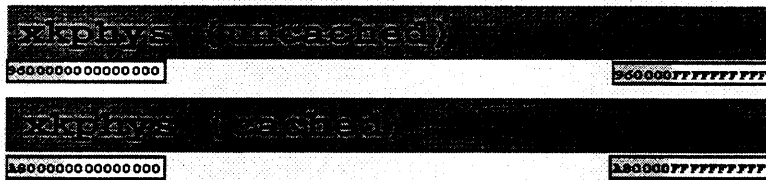
3-39.c

22jul1998

TR-IKI rev 0.7b SGI Proprietary

- the total number of "xkphys" pages actually in use is the sum of:
 - (pfdat "in-use" pages) + (total pages used for the kernel static data)
- an "xkphys" address that is within the bounds of the chip bit range, and is within the bounds of the range of actually configured memory, is still invalid, if the address refers to a physical page not currently in use
- "xkseg" and "xkuseg" addresses
 - must be "mapped" to be valid
 - this means that, at some point, a page of physical memory must have been allocated and matched with that xkseg or xkuseg virtual address, *and* an entry has been made in a table to reflect this (a "PDE" <Page Descriptor Entry> in a "PTE" <Page Table Entry> table)
 - the only exceptions are the two kernel "wired" TLB (xkseg) entries which are not referenced in the kernel's PTE table
 - an "xkseg" or "xkuseg" address that is within the bounds of the chip bit range, and is within the bounds of the software configuration limits for that kind of address, is still invalid if the address refers to a virtual page which does not have a valid PDE entry, that is, that virtual page has not yet been assigned a matching physical page of memory (again, with the exception of the special kernel "wired" entries).

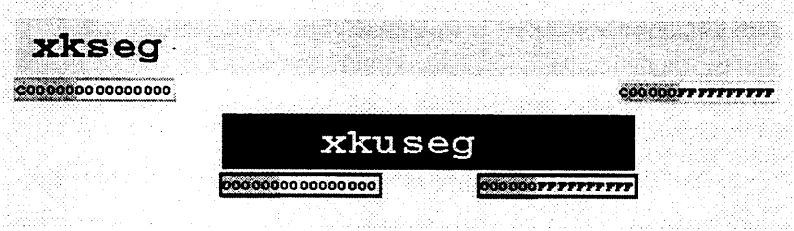
"Unmapped" Virtual Address Segment Types



Unmapped Addresses - xkphys

- xkphys virtual addresses are considered "unmapped", which means the last 40 bits of the word, bits 39:0, are treated as the reference to a specific physical page, or "PFN" (Physical Frame Number)

"Mapped" Virtual Address Segment Types

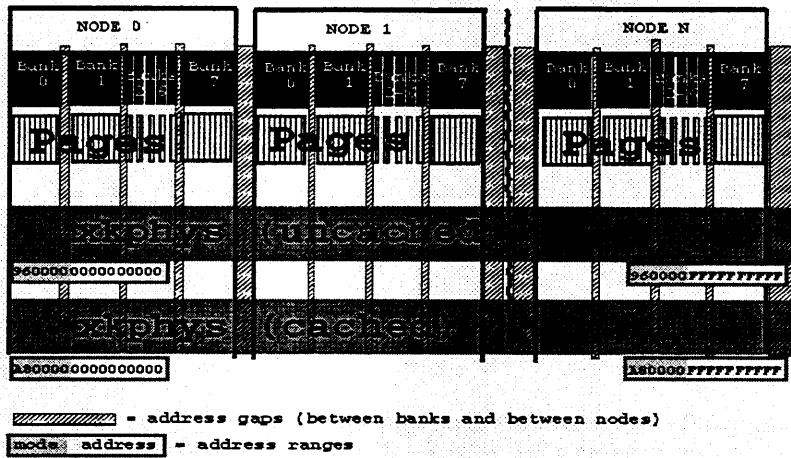


Mapped Addresses - xksege and xkuseg

- xksege and xkuseg virtual addresses are "mapped" addresses, which means the last 40 bits of the word, bits 39:0, must be translated to determine the matching physical page being referenced
- a page of physical memory must have been allocated and matched with that xksege or xkuseg virtual address, *and* an entry has been made in a table to reflect this (a "PDE" <Page Descriptor Entry> in a "PTE" <Page Table Entry> table)
- virtual-to-physical address translations that have already been calculated are stored in a CPU's TLB (Translation Lookaside Buffer)
- older TLB entries are overwritten eventually with newer address translations
- a CPU's TLB can be considered a cache of the most recently used virtual-to-physical address translations
- the kernel maintains two special TLB entries that are considered "wired"
 - they are always resident in the TLB
 - they are *not* referenced in the kernel's PTE table
 - unlike other virtual addresses
 - these two kernel (xksege) pages are allocated during system startup
 - the 32Mbytes of physical pages assigned to these two virtual kernel pages never change

- it is unnecessary to keep a table entry showing what physical pages are currently assigned to these virtual addresses

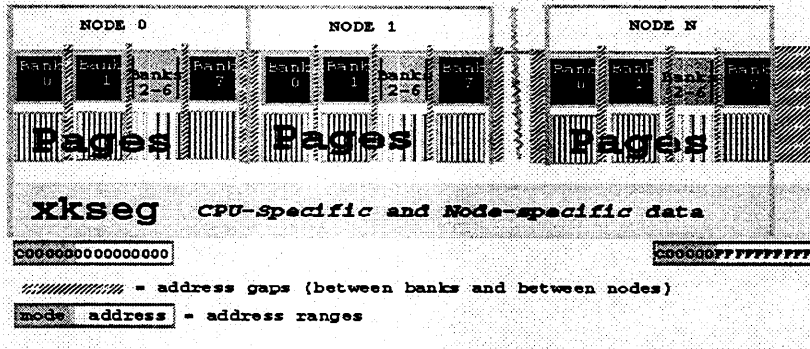
xkphys Memory Segments Diagram



xkphys Memory Segments - Detail

- 1:1 correspondence between xkphys virtual addresses and address range of physical memory, including "bad" (gaps, nonexistent) physical addresses
- "96" = value of first byte of *uncached* xkphys memory segment virtual address
- "A8" = value of first byte of *cached* xkphys memory segment virtual address
- bits 61:59 determine caching algorithm
- xkphys virtual addresses are considered "unmapped", which means the last 40 bits of the word, bits 39:0, are treated as the reference to a specific physical page, or "PFN" (Physical Frame Number)

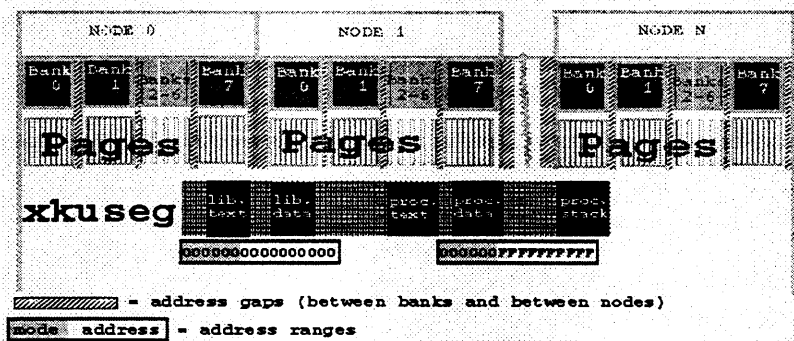
xkseg Memory Segment - Introductory Diagram



xkseg Memory Segment - Introduction

- Only a small subset of physical pages are actually used for xkseg type memory references
- xkseg virtual addresses are "mapped", that is, the last 40 bits of the word, bits 39:0, are *not* treated as a direct translation to a physical page of memory
- "C0" = value of first byte of xkseg memory segment virtual address
- xkseg used for CPU-specific and node-specific data (eg. , kernel tables, each node's copy of IRIX, etc.)

xkuseg Memory Segment - Introductory Diagram



xkuseg Memory Segment - Introduction

- the xkuseg virtual addresses are "mapped" addresses, that is, the last 40 bits of the word, bits 39:0, are *not* treated as a direct translation to a physical page of memory
- the xkuseg virtual address range covers *much less* than the total possible range of physical addresses
- the xkuseg memory segment is the size of *one* user process (the maximum permissible process size)
- each CPU uses the *entire* xkuseg memory segment to refer to *a single* user process address space
- each CPU refers to the *same* range of xkuseg virtual addresses to describe that CPU's currently connected process (eg, "the" 10th page of "the" currently connected process)
- the same *virtual* address maps to different *physical* pages for each CPU.
- "sparsely populated" - only a small subset of physical pages are actually used by a CPU referencing xkuseg virtual addresses

xkseg - Detail

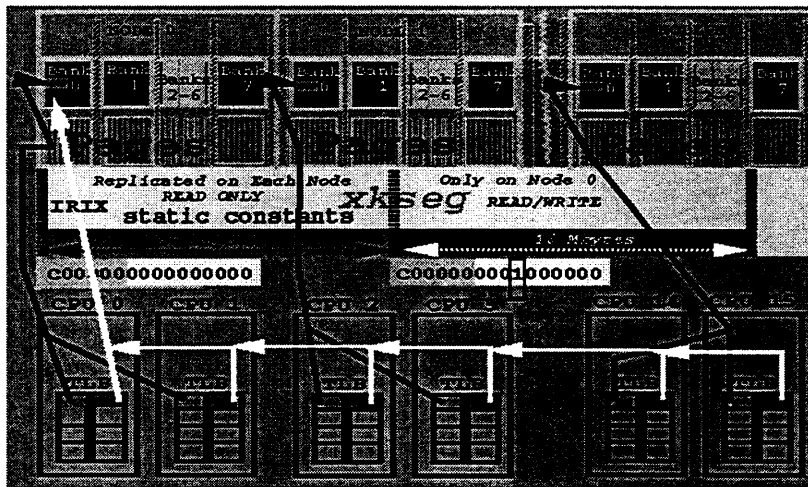
xkseg Virtual-to-Physical Address Translation - Diagram



xkseg Virtual Addresses Mapped through the TLB to Physical Addresses

1. A CPU is presented with an *xkseg* virtual address
2. The CPU examines its TLB to see if a virtual-to-physical address translation has already been calculated for the page containing the desired address (in this picture, this is the case. There are explanations later in this section for the more complicated cases where PTE tables must be examined to determine the translation.)
3. The same virtual address presented to different CPU's can be translated to **different physical addresses**

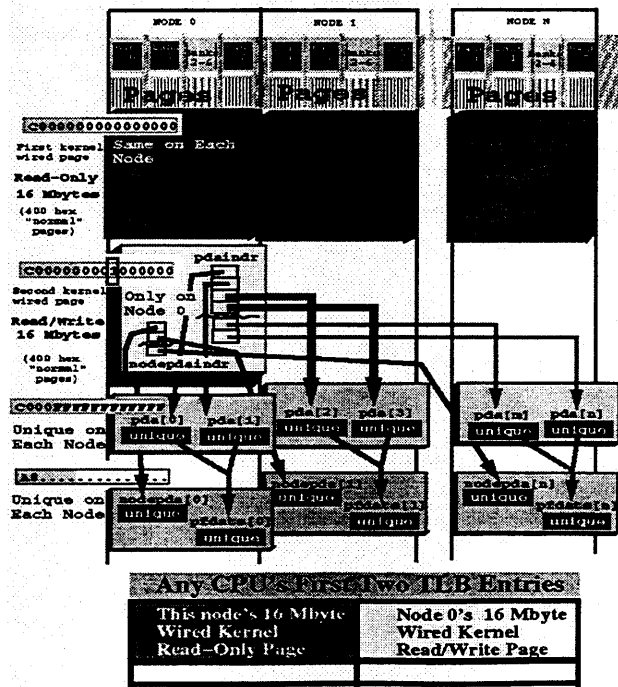
xkseg Wired Kernel TLB Entries - Diagram

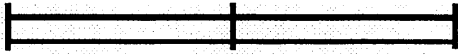


xkseg Wired Kernel TLB Entries

- TLB has first few entries "wired" (preset) and used for kernel references
- system default page size is 4Kbytes
- page size must be a multiple of 4Kbytes
- normal Cray Origin2000 page size is 16Kbytes
- page size is configurable
- special kernel page size set to 16Mbytes (= 400 hex pages of 'normal' pages of 16Kbytes each)
- each R10000 chip TLB register entry contains *two* virtual-to-physical address translations
- the first "wired" TLB entry contains references to two kernel-sized pages of 16Mbytes each
- a CPU referencing any virtual address within the first 32 megabytes of kernel memory (xkseg mapped "C0" prefix addresses) will always find a TLB entry mapping that virtual address to a physical page
- no delay to handle a "TLB miss"

Contents of xkseg Kernel Wired Entries





- the first wired kernel TLB entry
 - refers to virtual address (C0...) pointing to physical pages on that node
 - structures and data repeated on all nodes at same virtual addresses (different physical addresses)
- the second wired kernel TLB entry
 - refers to virtual address (C0...) pointing to physical pages on Node 0
 - some structures and data repeated on other nodes, but *not* at same virtual addresses (or physical addresses)
 - when these structures and data are referred to with xkphy addresses (A8... or 96...), the physical pages referenced are on the same node
- the pfdat table
 - one pfdat structure is used to manage every physical page (1:1) of memory except for
 - those physical pages used on each node to contain the kernel's 16MByte Read-Only data
 - those physical pages used on Node 0 to contain the kernel's 16MByte Read/Write data
 - only those pfdat entries reflecting the physical pages for that node are kept on that node
 - each node has multiple linked lists of collections of contiguous free pages and contiguous in-use pages on that node

Detail:

For CPU [4] (on Node 2) to locate an available, or "free", physical page of memory near CPU[1] (on Node 0), CPU[4] walks through the following structures and pointers:

1. CPU[4]
 - looks in CPU[4]'s pdaindr table for the location of CPU[1]'s PDA (the second entry)
 - CPU[1]'s PDA is located on Node 0
2. On Node 0, in CPU[1]'s PDA find the sub-structure "p_nodepda" (a structure called "nodepda_s")
3. In the p_nodepda structure is another sub-structure, "node_pg_data" ("pg_data_t")
4. The node_pg_data contains a sub-field, "pg_freelst" which points to a structure called "pg_free_s"

3-59.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

5. The pg_free_s structure contains a sub-structure
 - This sub-structure is made up of an array of "phead" structures "phead" (of type "phead_t")
6. Each phead structure itself contains an array of "ph_list" structures of type "plist_t")
7. The first field of the ph_list array is "pf_next", which is a linked list that points to the first set of free pfdats on that node

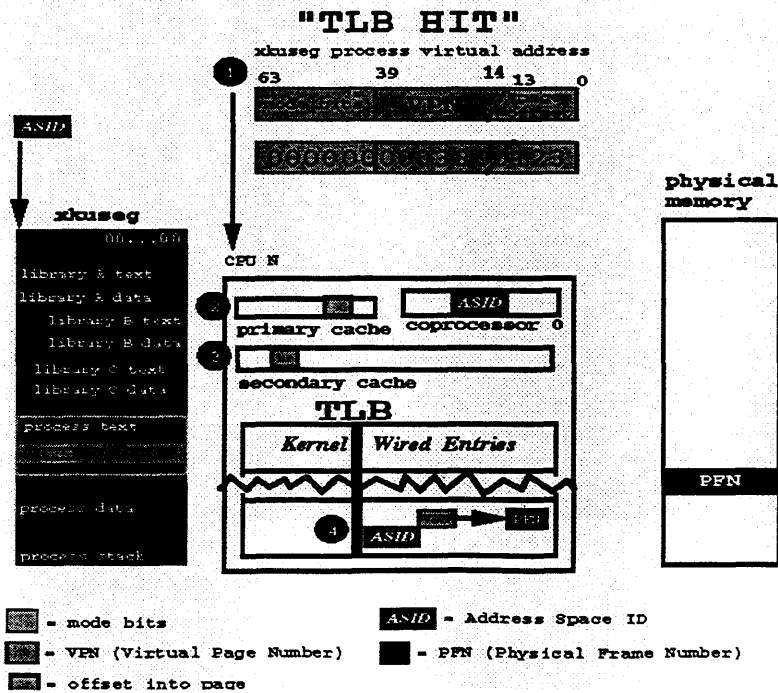
3-59.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

xkuseg - Detail

xkuseg "TLB Hit" - Diagram



User Addresses Are (Also) Mapped Through the TLB:

1. A user process xkuseg virtual address (00...) is presented to CPU

Depending on the architecture, steps 2-4 may be done sequentially, or simultaneously:

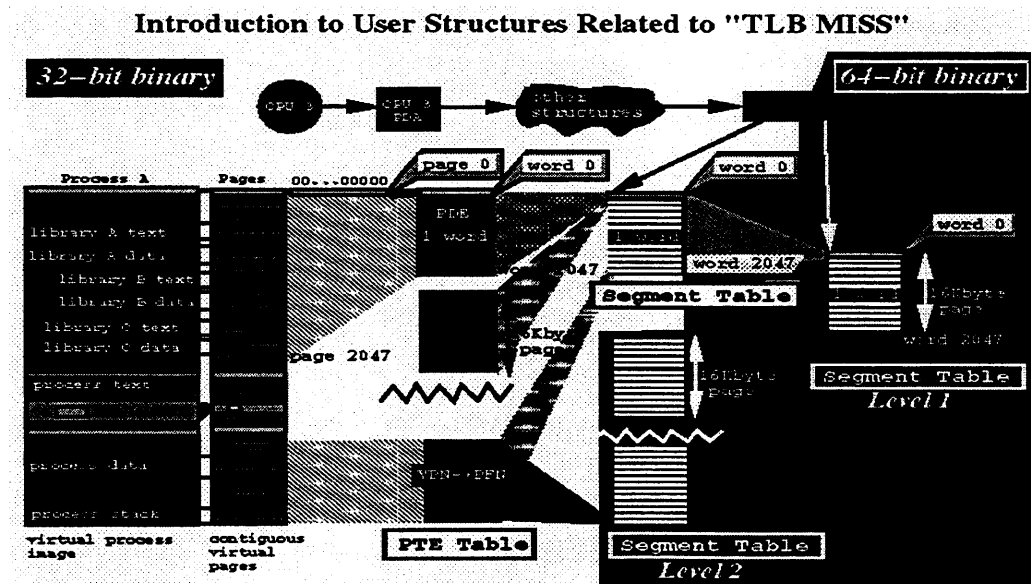
1. CPU looks in primary cache for the byte offset into the page
2. CPU looks in secondary cache for the byte offset into the page
3. CPU looks in the TLB for a matching combination of both the virtual page number (VPN) and the Address Space ID (ASID)

Address Space Identification (ASID) explanation

Each independent task, or process, has a separate address space, which is assigned a unique 8-bit Address Space Identifier (ASID). This identifier is stored with each TLB entry to distinguish between entries loaded for different processes. The ASID allows the processor to move from one process to another (that is, perform a "context switch") without having to invalidate TLB entries.

The processor's current ASID is stored in the low 8 bits of the EntryHi register in coprocessor 0. These bits are also used to load the ASID field of an entry during TLB refill.

The ASID field of each TLB entry is compared to the EntryHi register; if the ASIDs are equal (or if the entry is global, which means the entry is valid for all processes), this TLB entry may be used to translate virtual addresses.

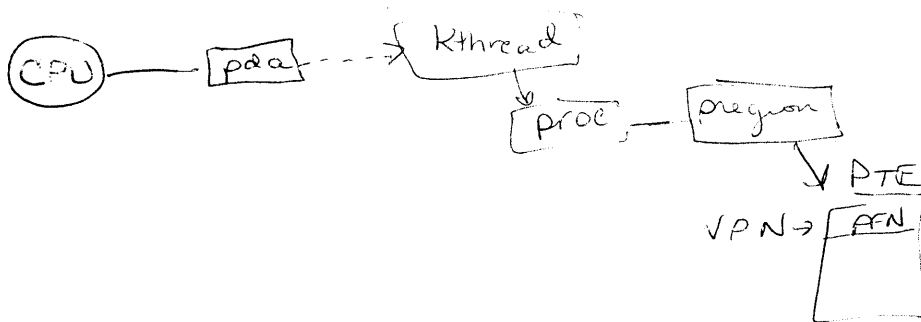


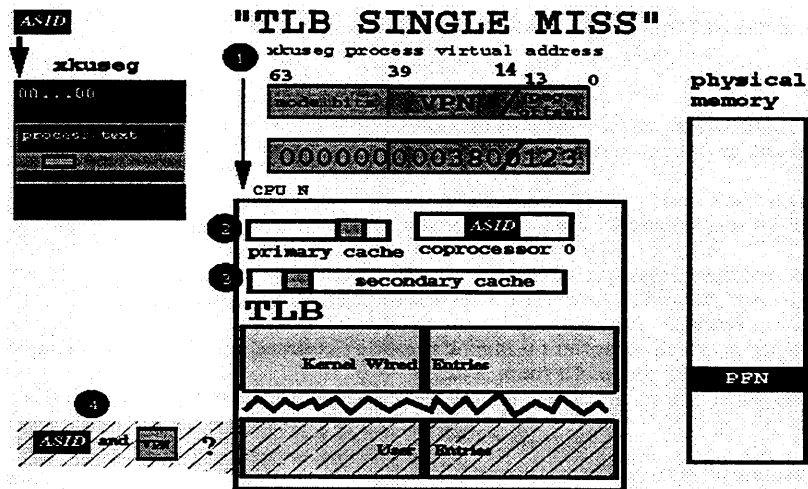
Introduction to User Structures Related to "TLB Miss" (TLB Exception)

- PDA (Private Data Area)
 - Each CPU has its own PDA.
 - Used by each CPU for many things, including to track what context this CPU is in, what thread this CPU is connected to, inter-CPU communication, and much more.
 - See "pda.h" for structure contents.
 - Always appears at the same virtual address in each processor
 - It is one page (4K), and the bottom 1024 bytes is used as about/idle stack.
- uthread structure
 - Each process has at least one uthread structure associated with it.
 - The uthread structure is the "starting point" for a CPU to refer to a process. Each CPU's PDA points to its currently connected thread (uthread structure).
 - The uthread structure contains, or points to, many important things, directly or indirectly, including associated vnodes, valid user pages, associated buffers, this process's stack, etc.
 - The uthread at 6.5 is the focal point for referring to a process in the way that the "proc" structure was the focal point in earlier releases of IRIX.
- Segment Table
 - The segment table is used to manage the user process image pages.
 - Each segment table entry is one word long.
 - One (16Kbyte) page of a segment table holds 2048 words
 - The uthread structure points to the first entry in the segment table
 - For 32-bit binaries
 - each one-word entry in the segment table points to the beginning of a page of user PDE entries.
 - one page of (one word each) segment table entries refers to 2048 consecutive pages of user PTE pages (each of which contains 2048 words, referring to 2048 user pages)
 - For 64-bit binaries
 - each one-word entry in the first-level segment table points to the beginning of a page of secondary-level segment table one-word entries.
 - Each one-word entry of the secondary-level segment is used in the same way as the segment table for

32-bit binaries, that is, each secondary-segment table word points to a page of PDE entries, and each PDE entry contains a reference to a user's virtual page (VPN), and the physical page (PFN) (if any) that has been mapped to it.

- PTE Table and PDE entries
 - A PDE is a one-word entry (in the PTE table) which refers to a page of a user process
 - A PDE contains both the reference to the user's virtual page number (VPN), and the mapping to the actual physical page of memory (PFN) the page is using, if any.
 - The contents of a PDE word are used to load a TLB entry.
 - The PDE words in a user's PTE table form a consecutive one-to-one correspondence with that user's virtual user (xkuseg) pages
 - One (16Kbyte) page of user PDE's holds 2048 words, and can therefore refer to 2048 virtual user pages
 - The PTE (Page Table Entry) and PDE (Page Descriptor Entry) structures are "unioned" in the code, that is, both names are used to refer to the same structure (most of the code refers to "pde"s, except for "irix/kern/ml/tlb.s", which refers to "pte"s)





TLB Single Miss

As before:

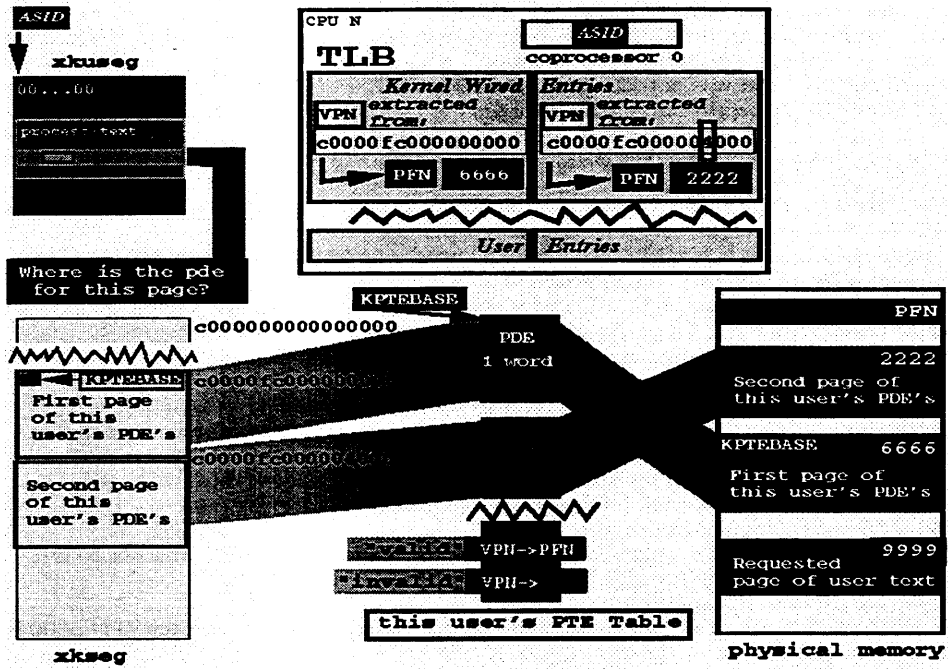
1. A user process xkuseg virtual address (00...) is presented to CPU

Depending on the architecture, steps 2-4 may be done sequentially, or simultaneously:

1. CPU looks in primary cache for the byte offset into the page
2. CPU looks in secondary cache for the byte offset into the page
3. CPU looks in the TLB for a matching combination of both the virtual page number (VPN) and the Address Space ID (ASID)

This time, however, this user's VPN is *not* found in the TLB.

Overview of Resolving a TLB Single Miss



Overview of Resolving a TLB Single Miss

- While this CPU is executing in user context:
 - The currently connected user process takes up all of the xkuseg memory segment, or "00..." range of addresses.
 - The VPN was extracted from the user's xkuseg ("00...") virtual address.
 - This VPN was presented to this CPU and was not found in this CPU's TLB.
 - This causes a "TLB Miss", or "TLB Exception" in the hardware.

This CPU will change context from user mode to kernel mode.

- While the CPU is executing in kernel context the CPU will begin to execute kernel code to do the following:
 - The kernel will find the base of this particular user's PTE table of PDE's.
 - The base of each user's PTE table is a "well-known kernel address" of ~~c000fc000000000~~, which is associated with a kernel variable named "KPTEBASE".
 - Each time a CPU is connected to a user proces, this same xkseg (c0...) virtual address is remapped to point to a *different* physical page in memory, where the currently connected user's PTE table begins.
 - The CPU will calculate the offset from the beginning of the PTE table, "KPTEBASE", that represents the PDE for the user virtual page we want.
 - Each PDE describes a virtual user page (VPN) which may, or may not, be valid.
 - A virtual user page (VPN) is "valid" if it is associated with an actual physical page of memory and that association has been written into the appropriate PDE for that VPN.
 - Each virtual user page (VPN) is represented by a one-word PDE entry.
 - We want the "VPNth word" from the beginning of the user's PTE table.
 - If we are looking for user virtual page 10:
 - then the tenth word of the PTE table will contain the information about whether VPN 10 is mapped to a physical memory page (PFN) or not.
 - If we are looking for user virtual page 3000:
 - then the 3000th word of the PTE table will contain the information about whether VPN 3000 is mapped to a physical memory page or not.
 - On a 64-bit architecture, the default page size is 16 Kbytes, or 2048 words.

utlbmiss()

Xcontext register

3-69

22jul1998

TR-IKI rev 0.7b SGI Proprietary

- Each page of the PTE table contains 2048 one-word-long PDE entries.
- The 3000th entry is located in the second page of PDE's.
- The second page of PDE's will have a *contiguous virtual address* ("c0...") after the first page of PDE's. The *physical address* probably is *not contiguous*. It is the contents of the physical page that we must examine to find the 3000th PDE.
- The result of this calculation, like all virtual addresses, will be an offset from the beginning of some page (page # + offset).
- The virtual address will be an xkseg ("c0...") address. These addresses are virtual "mapped" addresses, and must be looked up in the TLB, in order to determine what physical page actually holds the information.
- The CPU will extract the virtual page number (VPN) from the ("c0...") xkseg virtual address, and look in the TLB to find what physical page actually holds that set of user PDE entries.
- The CPU will examine that physical page of memory, find the offset from the beginning of that page that contains the PDE information that is needed, and will then load the contents of that PDE entry into the TLB.

The CPU will perform a context switch back to user mode.

- While this CPU is executing in user context:
 - The CPU will re-execute the original user instruction which caused the "TLB Miss".
 - This time, the virtual page number (VPN) and its associated physical page (PFN) are found in the TLB.
 - This is a "TLB Hit". The secondary and primary caches are loaded, and the needed byte is presented to the CPU. The instruction is executed.

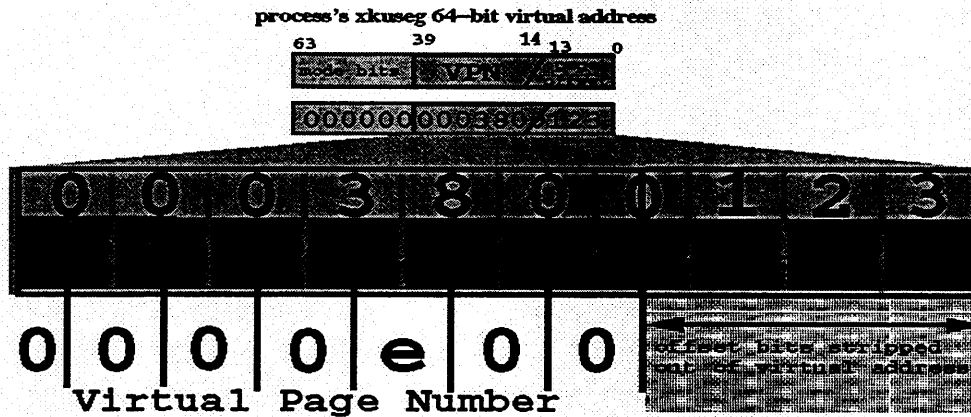
3-69.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Detail of Resolving a TLB Single Miss

Calculate the VPN to look for in the TLB



1. A user process is connected to a CPU. On a 64-bit architecture, among other things:
 - The TLB is loaded with a pair of PDE entries, which represent the first and second pages of the user's PTE table. Each of the two xkseg ("c0...") virtual addresses is mapped to its physical page in its respective TLB entry.
2. The CPU is presented with a virtual user xkseg ("00...") address of "0000000003800123".
3. The CPU looks in the primary and secondary caches and does not find this **byte offset** of this page. These are "cache misses".

TR-IKI rev 0.7b SGI Proprietary

22jul1998

3-71

4. The CPU needs to look in the TLB for the virtual **page number** (VPN) (and matching ASID).
 - The virtual page number must be calculated from the 64 bit word.
 - The 64-bit word contains both mode bits and address bits.
 - The address bits contain both the page number and the offset into the page.
 - The offset is represented by bits 13:0.
 - Bit 13 falls in the middle of one of the 16 hex digits it takes to represent a 64-bit word.
 - Bits 39:14 contain the virtual page number. Examining these bits shows the VPN is "e00".
 5. The CPU looks in the TLB for VPN "e00" (and the ASID that matches this user address space).
- In this case, the user's VPN "e00" is not found in the TLB.
6. This causes a "TLB Miss" (slang), or (the more technically accurate name) "TLB Exception" in the hardware.
 7. The CPU does a context switch to kernel mode and enters kernel code to handle a TLB Exception.
 8. Kernel code is performed to:
 - find the beginning of the kernel structure with this user's PDE's.
 - "KPTEBASE", xkseg address "c0000fc000000000", points to this.
 - find the particular one-word PDE entry that contains information about user VPN "e00" and what physical page (PFN) has been assigned to it.
 - 1. Hex math is performed to find the byte offset from "KPTEBASE" (the beginning of the user's PTE table) which contains the PDE word for the requested user VPN.
 - A PDE entry is only one word (8 bytes of 8 bits each=64 bits) long.
 - A 16Kbyte page of PDE's contains 2048 PDE word entries.
 - We want that page of PDE entries that has the "e00"th word.
 - (VPN) * 8 = the first byte of the 8-byte PDE word we want.
 - (0xe00) * 8 = 0x7000
 - The first byte of the PDE word with the information about user VPN e00 starts 7000 bytes from the beginning of the PTE table.

See next illustration.

TR-IKI rev 0.7b SGI Proprietary

22jul1998

3-71.a

Hex math formula to calculate byte offset from KPTEBASE:

$$\frac{\text{VPN (the page number)} \times 8 \text{ (the number of bytes per word)}}{0xN \text{ (= the hex number of bytes offset from the start of the user's PDE table)}}$$

Examples:

<p>if VPN=0</p> $\frac{0}{x \ 8}$ <p>= byte 0 (the beginning of the first PDE, word 0, bytes 0-7)</p>	<p>if VPN=1</p> $\frac{1}{x \ 8}$ <p>= byte 8 (the second PDE, word 1, bytes 8-15)</p>	<p>if VPN=e00</p> $\frac{e00}{x \ 8}$ <p>= byte 7000 (the 7000th PDE word)</p>
---	--	--

Hex math shows which page of PDE's contains the 7000th byte.

See next illustration.

Hex math formula to calculate which page of PDE's contains the first byte of the desired PDE.

$$\frac{0xN \text{ (= the hex number of bytes offset from the start of the user's PDE table)}}{0x4000 \text{ (hex) bytes per 16Kbyte page}} = \text{Page number}$$

(any remainder is the offset into that page)

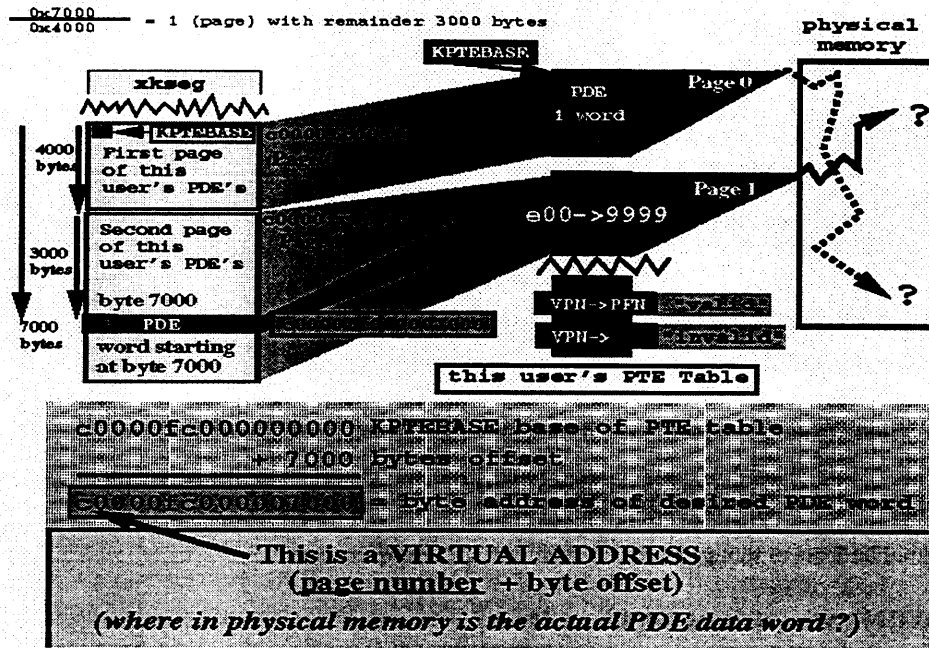
Examples:

<p>(for user VPN = 0) for byte=0</p> $\frac{0x0}{0x4000}$ <p>= page 0 remainder = 0</p> <p>The desired PDE is in PTE page 0, starting at byte 0</p>	<p>(for user VPN = 1) for byte=8</p> $\frac{0x8}{0x4000}$ <p>= page 0 remainder = 8</p> <p>The desired PDE is in PTE page 0, starting at byte 8</p>	<p>(for user VPN = e00) for byte=7000</p> $\frac{0x7000}{0x4000}$ <p>= page 1 remainder = 3000</p> <p>The desired PDE is in PTE page 1, starting at byte 3000</p>
---	---	---

The PDE that describes user VPN e00 starts 7000 bytes from KPTEBASE, the beginning of this user's PTE table. This puts it offset 3000 bytes into the second page of user PDE's (page 1).

1. The kernel calculates the virtual address of the 7000th byte.

See next illustration.



(where in physical memory is the page containing that word?)

The virtual page number (VPN) must be looked up in the TLB

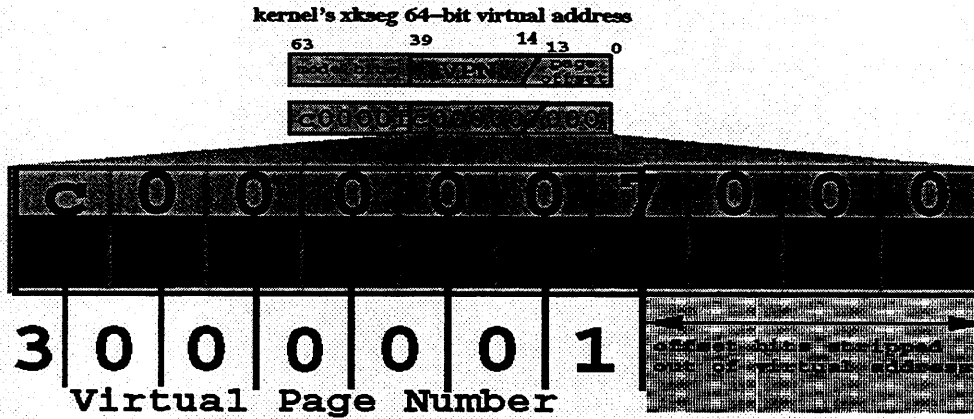
The address of the 7000th byte is an kxseg ("c0...") virtual address of "c0000fc0000007000".

This kind of address is a "mapped" address, just like the user's kxseg virtual address.

1. The VPN for this kernel virtual address has to be calculated from bits 39:14 in exactly the same way the user's VPN was determined.
 - o Then that VPN is looked for in the TLB.
 - o The TLB entry will match the virtual page number (VPN) with the actual physical page of memory where this page of user PDE's starts.

See next illustration.

Calculate the VPN to look for in the TLB

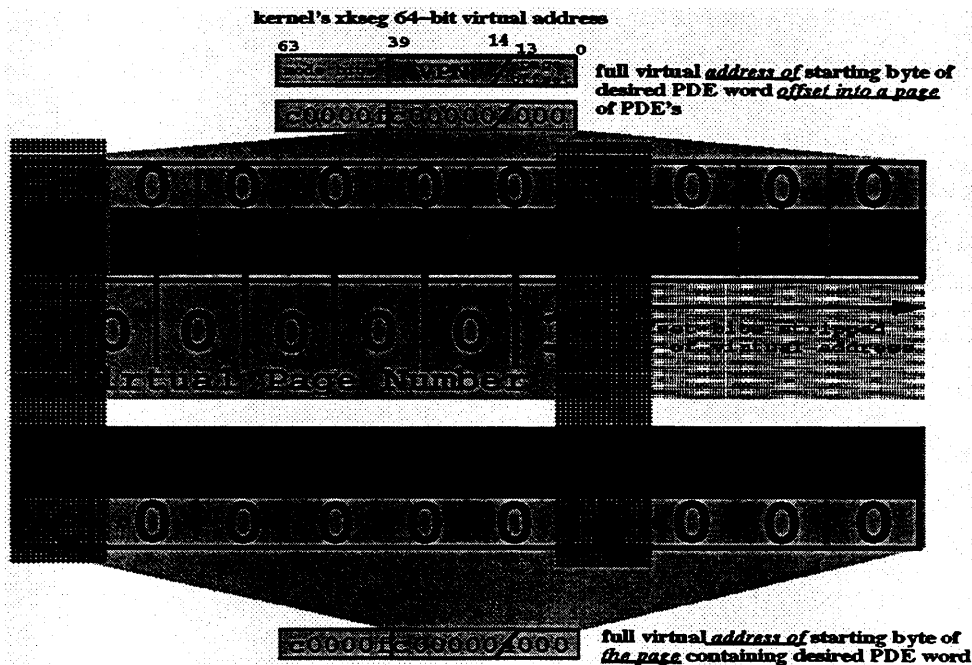


The kernel xkseg ("c0...") virtual address of the PDE data word we want is "c0000fc00007000".

That word is offset in a page full of user PDE words. When the offset bits are stripped out of the virtual address, the virtual page number where that page of PDE's starts is "300001".

The full kernel xkseg ("c0...") virtual address of that page can be calculated, as shown below.

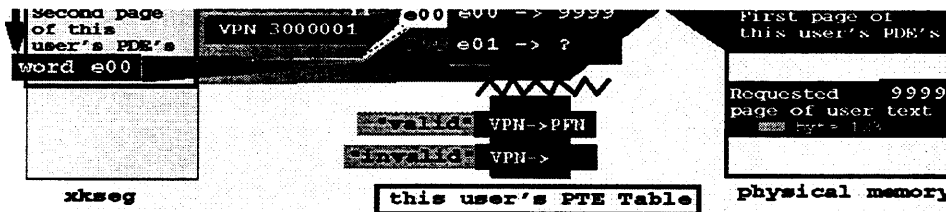
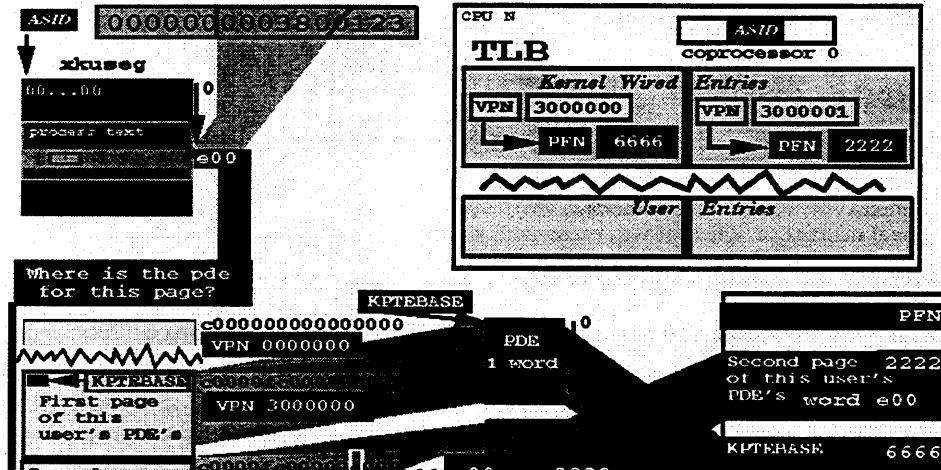
Translation of VPN back into full virtual address



1. The CPU, still in kernel context, looks in the kernel TLB entries for virtual page number (VPN) "300001" of the

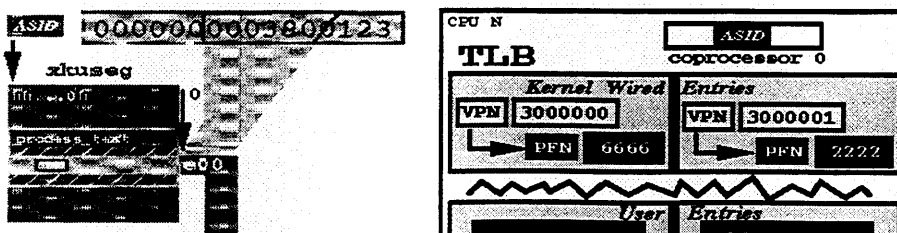
xkseg memory segment pages.

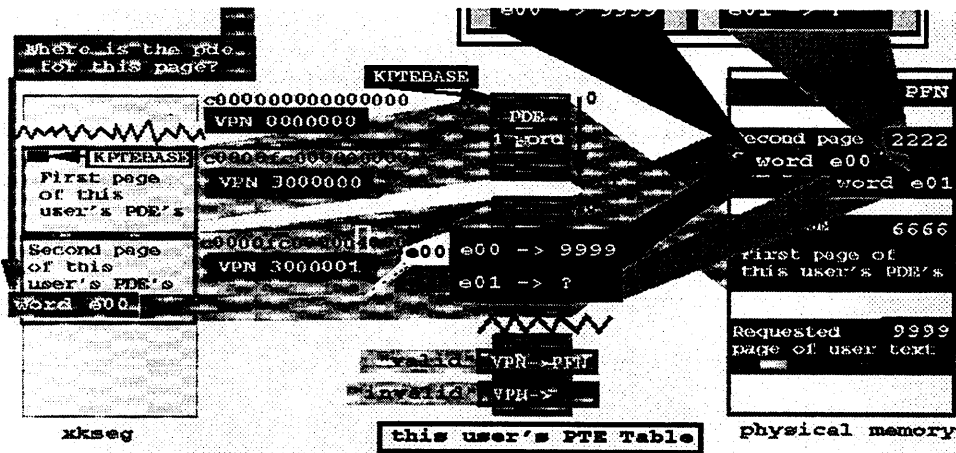
- The (kernel xkseg memory segment) virtual byte address "c000fc000007000" exists within a (kernel xkseg memory segment) virtual page that starts at byte address "c000fc000004000".
- Of the 64 bits of virtual address, bits 39:14 constitute the virtual page number (VPN). The VPN for both "c000fc000004000" (the start of the page) and "c000fc000007000" (a word within the page) is the same, ie, "3000001".
- The VPN is what is used to look in the TLB entries, in order to find where in physical memory the entire page of data has been written.
- In the illustration below, VPN "3000001" has been written to physical memory page (PFN) "2222" (a number chosen arbitrarily for this example).



1. The kernel finds VPN 300001 mapped to physical page (PFN) 2222 in the TLB.
2. The kernel goes to physical page 2222 of memory, and then to the proper offset of 3000 more bytes into that page (a distance of e00 words from KPTEBASE), and is now positioned at the first byte of the PDE word that represents information about user VPN e00 and whether it has been assigned an actual physical page of memory or not.
 - In the illustration above, KPTEBASE[e00] shows that user virtual page number (VPN) e00 has been written to physical memory page (PFN) "9999" (a number chosen arbitrarily for this example).
3. The kernel now loads this PDE entry (and on this architecture, the next contiguous PDE entry as well) into the TLB for this CPU.

See next illustration.





1. The CPU now does a context switch back to user mode.
1. The CPU re-executes the original instruction where it was presented with the xkuseg ("00...") user virtual address "000000003800123".
2. As before, the CPU looks in the primary and secondary caches for the desired byte, and as before, still does not find it.
3. As before, the CPU does hex math to extract the virtual page number (VPN) from the 64-bit xkuseg address, and determines that the VPN is "e00".
4. As before, the CPU looks in the TLB to see if VPN "e00" for this user's ASID exists as a valid TLB entry.
5. This time, the CPU does find this user's VPN of "e00" in the TLB, we have a "TLB Hit", and the CPU presents the physical page "frame" number (PFN), and offset into the page, to the HUB, and requests that it find the page and

3-71.j

22jul1998

TR-IKI rev 0.7b SGI Proprietary

return enough bytes to load both the primary and secondary caches, as well as providing the CPU with the particular byte desired.

3-71.k

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Detail of Resolving a TLB Double Miss

The sequence of events which lead up to a "TLB Double Miss", are very similar to the events which cause a "Single TLB Miss", until the kernel tries to look in the TLB for the VPN containing the user's PDE entries.

In a TLB "Double Miss", not only is the user's VPN not referenced in the TLB (the "e00" in the previous example), but, after the CPU does a context switch to kernel mode, the kernel can't find the TLB entry for the page of user PDE's that refers to the VPN the user wanted either (the "3000001" in the previous example).

When the CPU in user context can't find a requested VPN, that's a single miss. After the context switch to handle the single miss, the double miss occurs when the kernel discovers the user is trying to reference a VPN contained in a page of PDE's that aren't referenced in the kernel's TLB entries.

A TLB "Double Miss" Starts out the Same as a TLB Single Miss:

1. A user process is connected to a CPU. Among other things, this process is assigned an Address Space ID (ASID).

While the CPU is in user context:

1. a user 64-bit hex virtual address is presented to the CPU.
 - o Let's use the same example as before, and use the xkuseg ("00...") virtual address "000000003800123".
2. The CPU looks in the primary and secondary caches for the byte address requested.
3. The CPU does hex math to determine the Virtual Page Number ("VPN") contained in bits 39:14 of the 64-bit virtual address.
 - o In this example, the VPN is "e00".
4. The CPU looks in its TLB for a user VPN "e00" that matches this user's Address Space ID (ASID).
 - o A valid TLB entry will match, or "map", a virtual page number (VPN) with a physical page of memory (Physical Frame Number, or "PFN").
 - o If the VPN and ASID pair are found in the TLB, this is a "TLB Hit", and the CPU continues to process in user

- context.
- o If the VPN and ASID pair are not found in the TLB, this is a "TLB Single Miss", and this causes a hardware exception. This results in the CPU performing a context switch to kernel context, and beginning to execute kernel code, although the CPU is still considered "connected" to the user.

Additional Information About the Single Miss:

At this point the "EXL" bit in the status register of coprocessor 0 is set to indicate we have had one TLB Miss already.

For 32-bit binaries the kernel begins to execute at a "vector" or entry point called "UT_VEC", for a "TLB Exception".

For 64-bit binaries the kernel begins to execute at a "vector" or entry point called "XUT_VEC", for an "Extended TLB Exception".

While the CPU is in kernel context:

1. Hex math is done to calculate the offset from the beginning of this user's PTE table, which holds the word of data that describes what physical page of memory (PFN), if any, is being used by the requested VPN of, in our example, "e00".
 - o Each user process has a Page Table Entry (PTE) table made up of as many pages as are necessary to describe the range of xkuseg virtual addresses for a user process.
 - o Each PTE table entry, or "Page Descriptor Entry" (PDE) is one word long. Each PDE describes one virtual page (VPN), and which physical page (PFN), if any, is being used to contain the contents of this virtual page.
 - o On a system with a 16Kbyte page-size, this means that a single page of PDE's contains 2048 words and can,

therefore, contain the information to map 2048 user VPN's to physical pages (PFN's).

- The PTE table starts at a "well-known kernel (virtual) address" of "c0000fc000000000", which is pointed to by "KPTEBASE".
- Every CPU uses the same virtual address for the beginning of its currently connected user process's PTE table. These virtual addresses are, however, mapped to different physical pages in memory for each CPU's process or thread.
- Since one PDE represents the information for one user page, "KPTEBASE[VPN words] contains the PDE information for any given VPN.

The user's PTE table entries are referenced with xkseg kernel mapped ("c0...") virtual addresses. Mapped addresses must be looked up in the TLB to determine what physical page holds the referenced data. The TLB is examined for the value of the VPN extracted from bits 39:14 of the 4-bit virtual address. In our example, the VPN for the appropriate page of user PDE entries was VPN 300001.

The kernel enters a simple routine named "utlbmiss", which does simple math to calculate which page of user PDE words contains the PDE word that matches the VPN the user is interested in.

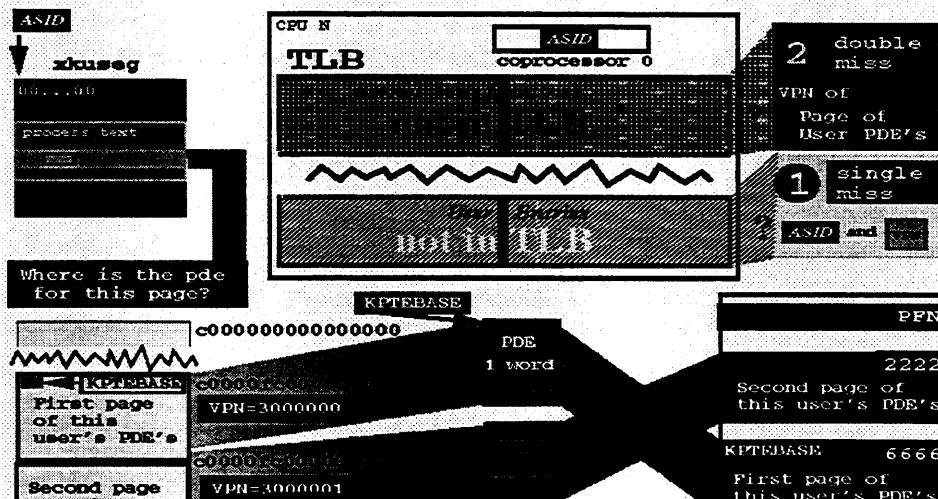
Assuming a valid PDE entry exists, one of the user TLB registers is selected at random to load with the value of "KPTEBASE[VPN]". Because there is a 1:1 correspondence between PDE words and user VPN pages, this formula loads the appropriate PDE word for the use VPN.

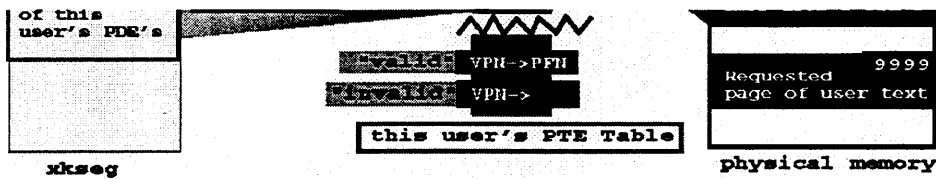
Where a TLB "Double Miss" is Different:

1. At this point, for a "TLB Single Miss", the CPU would find VPN 300001 in the kernel TLB entries. This TLB

entry would reference the PFN containing a set of 2048 PDE entries, each containing the VPN-->PFN mapping information for a single user page. The CPU would offset from the beginning of the physical page to the appropriate PDE entry, and load its contents. Also, due to the machine architecture in our example (ie, a single TLB register holds two PDE entries), the contents of the next physically contiguous PDE word, would also be loaded into the TLB. These two entries represent contiguous virtual pages, but the matching physical pages are probably not contiguous.

However, for a "TLB Double Miss", when the CPU looks in the TLB for VPN 300001, the kernel does not find this VPN either. This results in a second "TLB Miss", this time due to actions of the kernel, not the user process.

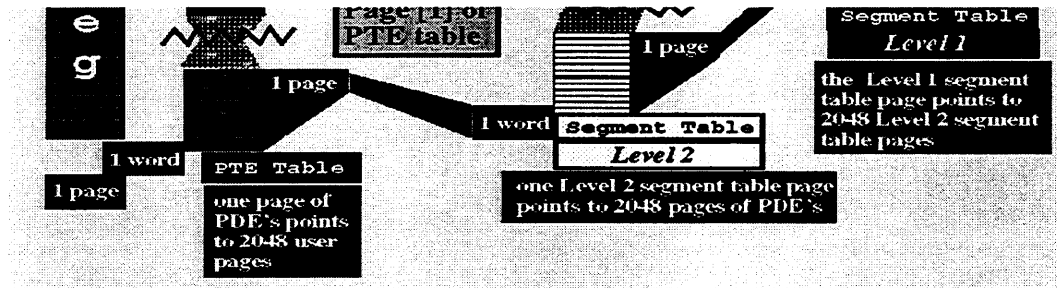
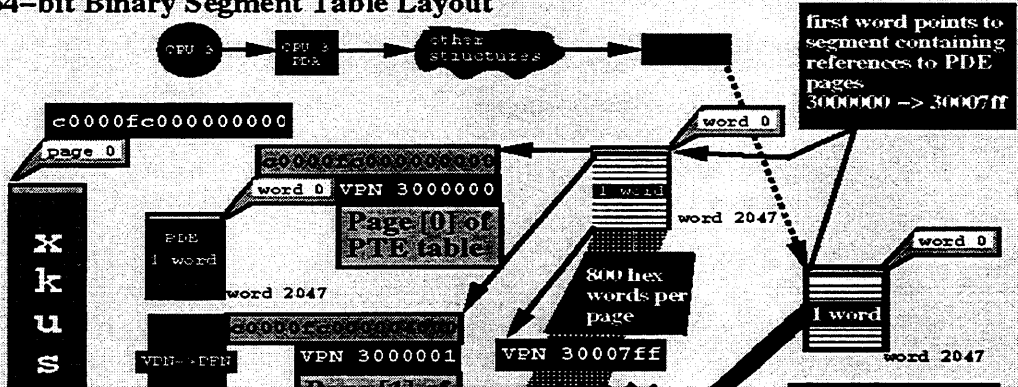


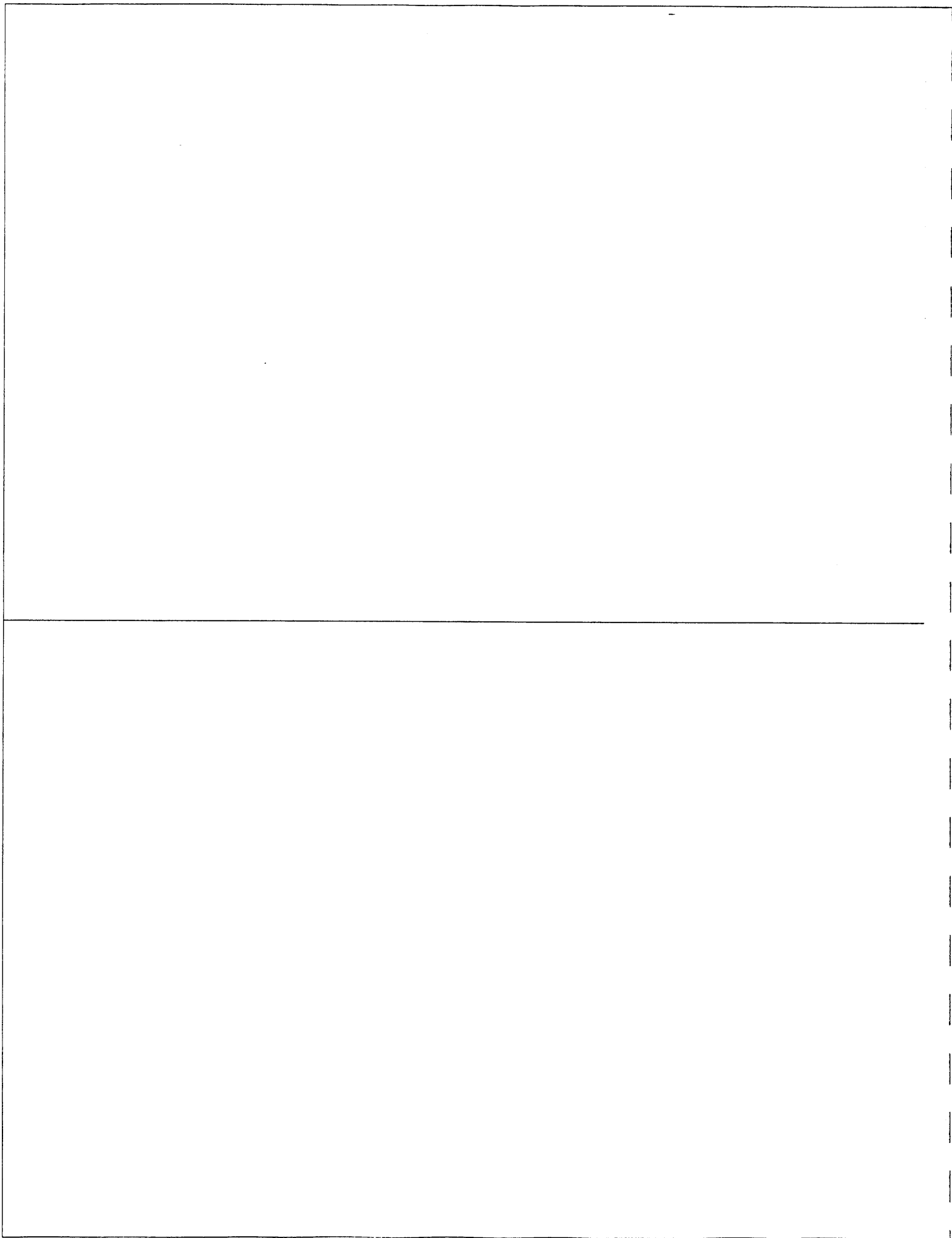


1. In the event of a "TLB Double Miss", the CPU can follow the pointers starting with its own PDA, which eventually lead to the segment tables which describe the pages of the currently connected user's PDE words.

Some of these structures were shown earlier, and some more detail is added in the diagram below. There is more detail on the structures in this chain later in a later section.

64-bit Binary Segment Table Layout





Module 4: Kernel Source Tree

Kernel Source Tree

This section provides an overview of the organization and location of operating system source code, with an emphasis on kernel code, and tools for browsing it.

By the end of this section, the student should be able to:

- Locate operating system release project web pages
- Locate operating system and kernel source code files
- Explain the kernel source tree subdirectory contents
- Explain the difference between ".h" files and ".c" files
- Determine if a ".c" file includes a specific ".h" file
- Describe the system logs, subdirectories, and files in `/var/adm`
- Describe tools to examine system activity
- Describe tools to examine a system dump
- Use the `cscope` tool to find operating system and kernel source code files
- Use the `uname` command to determine what software a system is running
- Use the `versions` command to determine what software a system is running

Related On-Line Materials

Additional information is available in the "Source and Object code maintenance" lesson, which provides detailed information about:

- Operating system release project web pages
- The location of operating system code source files
- The location of operating system code binary files
- How to track changes to the operating system source tree
- How to track changes to the operating system object code
- The location of kernel source code and tools to examine it
- Determining hardware and software system status with various tools and commands

That information is available as part of the selection of IRIX Class Materials located at:
<http://www.tng.cray.com/~mix/irix.html#Class-Materials> .

The lesson itself, and its Table of Contents, are located at:
http://www.tng.cray.com/~mix/protect/irix/manual/current/object_code.html .

Operating System Release Project Web Pages

IRIX 6.4.x ("ficus") Project Web Page:

http://info.engr.sgi.com/projects/bonnie_proj/ficus/isms/status/

IRIX 6.5 ("kudzu") Project Web Page:

http://info.engr.sgi.com/projects/bonnie_proj/kudzu/isms/status

Project Web Page information includes:

- Description of platforms supported
- Release milestones status
- Bug status ("showstoppers", summary reports, etc.)
- ISM ("Independent Software Module") owner and status
- Source and Build Information (source tree, "Build Meister", etc.)
- Related news groups, news letters
- Features list

There is a tremendous amount of valuable information contained on each operating system release's web page, including the location of important source code directories.

Source Code Location

The official location of source code is on "bonnie" in the "/proj" directory.

Once inside the SGI/CRAY firewall, there are two methods of getting to "bonnie" and various source trees:

```
telnet bonnie.engr.sgi.com
Trying 192.26.80.202...
IRIX (bonnie)
login: guest
```

(no password necessary)

```
cd /proj
```

or:

```
cd /hosts/bonnie.engr.sgi.com/proj
```

Base Source Code Naming Convention Explanation

Source code is not assigned a release number until it is close to being released. Once released, no further changes are made to that code directory tree.

IRIX 6.4 (which had the in-house name "ficus") has been released; "kudzu" has not, but will be released as IRIX 6.5 .

The "irix.65se" directory contains the most relevant source code for the Irix 6.5 *base* release for the Cray Origin2000 systems.

```
bonnie 6% cd /hosts/bonnie.engr.sgi.com/proj; ls -la
drwxrwxr-x 3 root sys  57 Dec 29 10:10 kudzu
drwxr-xr-x 4 root sys 4096 Dec 26 16:58 irix6.5
drwxr-xr-x 5 root sys  52 Jan 22 12:58 irix6.5-features
drwxr-xr-x 3 root sys  51 Jan  5 14:41 irix6.5-unbundled
drwxr-xr-x 3 root sys  57 Dec 29 10:19 irix6.5f
drwxr-xr-x 3 root sys  22 Jan 16 17:07 irix6.5m
drwxr-xr-x 3 root sys  51 Jan 19 03:10 irix6.5se
lrwxr-xr-x 1 root sys  14 Feb 11 1997 ficus -> irix6.4-s2mp+o
drwxr-xr-x 3 root sys  56 Dec 14 1996 irix6.4-s2mp+o
drwxr-xr-x 3 root sys  36 Oct 29 1996 irix6.4-ssg
drwxr-xr-x 3 root sys  22 Nov 18 1996 irix6.4-ssg-unbundled
```

Other directorie names are explained below.

The original IRIX 6.4 release can be found in the "irix6.4-ssg" directory. The "ssg" suffix refers to the " Scalable Systems Group". This release ran only on the high end "s2mp" (Scalable Symmetric Multi-Processor) architectures. This release has been superceded by the "irix6.4-s2mp+o" release.

The "irix6.4-ssg-unbundled" directory was never used, is empty, and can be ignored.

The code to support the low end "Octane" architecture was added in the "irix6.4-s2mp+o" version of the source, which is actually the IRIX 6.4.1 release.

The project leader for each release chooses the naming conventions, such as references to "ssg", "+o", or "unbundled". Do not rely on a consistent naming convention for released systems. All such conventions as naming and location are totally up to the manager of the group.

The original in-house name for the IRIX 6.4 release was "ficus". Note that the "ficus" directory has been symbolically linked to "irix6.4-s2mp+o". The pre-release in-house name will always be linked to the most relevant base release file name, once the operating system is released.

The all platform release of IRIX 6.5 can be found in the "irix6.5" set of subdirectories.

The "irix6.5-unbundled" subdirectories are for products, like the compilers, which do not ship with the kernel. These may have their own release cycles, and may be optional software.

The subdirectory named "irix6.5m" is for maintenance on the 6.5 release. This is where fixes for low priority bugs are checked in (problems not significant enough to hold up the release).

Where Is the Most Recent Version of the Source Code for the Upcoming IRIX 6.5 (kudzu) Release?

Source code for upcoming IRIX 6.5 release

- Always buildable
- Always viewable
- Changing constantly
- Being tested constantly
- Kept on "bonnie"

Source code is kept in a continuously releasable state until it is released. Changes and corrections are applied directly to the source code in the development tree for the release.

The source code for "kudzu", which will probably be released as "IRIX 6.5", is kept in:

`/hosts/bonnie.engr.sgi.com/proj/irix6.5/isms`

(`/hosts/bonnie.engr.sgi.com/proj/kudzu/isms` is outdated)

NOTE: "isms" directories should be:

- Independent bodies of code
- The most recent appropriate software module for the release... but they may not be !

WARNING: The acronym "isms" stands for "Independent Software ModuleS". These are supposed to be large bodies of independent code, which can be built independently, but there may be some interconnections between them anyway (e.g., all the graphics code, and all the man pages, and all the IRIX kernel do have interdependencies with each other). Any "isms" directory should contain "all" the source code for that release, however, compilers have their own separate release schedule and are not included in the same "isms" tree. Compiler versions can be found in subdirectories under:

TR-IKI rev 0.7b SGI Proprietary

22jul1998

4-7

`/hosts/bonnie.engr.sgi.com/isms/cmplrs.src`

(note: there is an Eagan cscope database of development IRIX 6.5 source, and there are IRIX 6.4 and 6.5 cscope databases on bonnie.engr.sgi.com in /cscope .)

Where Is the Most Recent Version of the Source Code for the Current IRIX 6.4 (ficus) Release?

There isn't one.

There is no policy for maintaining a recent *viewable* version of the *source* code for the current release.

The *base* source code for the IRIX 6.4 release is kept on "bonnie", in the directory:

```
/hosts/bonnie.engr.sgi.com/proj/irix6.4-smp+o .
```

The code in this set of subdirectories is readable, buildable source.

There is no policy for maintaining a current *viewable* version of the source code for any released operating system. Operating systems are released to customers in binary form (object code).

Once an operating system is released, it is no longer kept in a continuously releasable (buildable) state.

(note: there is an Eagan cscope database of released IRIX 6.4 source)

4-8

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Summary: Location of Operating System Source Code Base Release, Current Release, and Upcoming Release

- Source code for IRIX 6.4 base release (no patches)
 - Viewable
 - Buildable
 - Kept on "bonnie"
 - /hosts/bonnie.engr.sgi.com/proj/irix6.4-s2mp+o/isms
(/hosts/bonnie.engr.sgi.com/proj/ficus/isms is linked to the above)
- Source code for IRIX 6.4 base release with released patches applied
 - Viewable
 - *KERNEL* source *MIGHT* be buildable
 - *ALMOST all released patches applied*
 - No patches in test mode
 - Kept on "bonnie"
 - /hosts/bonnie.engr.sgi.com/proj/irix6.4_patched/isms
- Source code for IRIX 6.5 development
 - Viewable
 - buildable
 - Kept on "bonnie"
 - /hosts/bonnie.engr.sgi.com/proj/irix6.5/isms

(note: Eagan cscope databases of all three trees)

The source code for the *base* release will always be viewable and buildable. It will be located on "bonnie.engr.sgi.com" in the "/proj" directory, under a name which includes the release number (e.g., "irix6.4-s2mp+o").

There is no policy to maintain a viewable, buildable version of the current release with all or most currently released patches applied.

4-9

22jul1998

TR-IKI rev 0.7b SGI Proprietary

NOTE: One of the developers has taken it upon himself to create a *buildable* version of the latest *kernel*, with *resolved* patches applied. Whenever a new patch is inserted into the "irix6.4_patched/isms/irix/kern/*" tree, this developer will, himself, apply a new patch which will result in the kernel source tree being a buildable and "vi-able" version of the kernel, with most of the released patches (remember the time lag before a released patch shows up in the source tree) compiled in (that is, patch dependencies and conflicts will be resolved).

In short, what you find in the "irix6.4_patched/isms/irix/kern" subdirectories will toggle between two states. It will always be something that "vi" can examine. However, the code will be a buildable version of the kernel source, with *most* of the recently released patches, only until the build team drops in a new patch or patches. Then the code will be an unbuildable version of the kernel source, with *most* of the released patches, and with possible patch conflicts and dependencies. When this condition is noticed, the developer will apply yet another patch to toggle the code back to a buildable, resolved state.

Kernel Source Tree Location

All source files are on "bonnie.engr.sgi.com".

The main kernel source directories are located in:

```
/hosts/bonnie.engr.sgi.com/proj/release-name/isms/irix/kern
```

IRIX kernel source code for the IRIX 6.4 release is located in:

```
/hosts/bonnie.engr.sgi.com/proj/irix6.4-s2mp+o/isms/irix/kern
```

IRIX kernel source code for the kudzu (IRIX 6.5) release is located in:

```
/hosts/bonnie.engr.sgi.com/proj/irix6.5/isms/irix/kern
```

All source code can be found on the "bonnie.engr.sgi.com" host, mostly in an "isms" directory of Independent Software Modules, that is, code (mostly) logically distinct from other code. And most of the "interesting" operating system code is in the subdirectories under "isms/irix/kern".

The IRIX kernel source code for the kudzu (IRIX 6.5) release for the Cray Origin2000 architecture is located in:

```
/hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix/kern
```

Most of the kernel code is in a "/proj" subdirectory related to the appropriate release name, each of which has its own "isms" subdirectory. In each release's "isms" directory of independent code modules, there is an "irix" directory, which leads to a "kern" directory. Here is where the kernel code can be found.

Note: Many of the files and directories have been symbolically linked. Do not be confused if the pathway you "cd" to does not resemble your "pwd" output. The shell you use makes a difference. If you are using *csh*, then you could encounter something like this:

```
cd /hosts/bonnie.engr/proj/irix6.4-s2mp+o/isms/irix/kern ; pwd
```

/hosts/bonnie.engr/disks/xlv2/ficus/irix_ficus/kern

Using *ksh* rather than *csk* will avoid this confusion.

4-10.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Kernel Source Tree Contents

Below is an explanation of the files and directories where various parts of the kernel source tree, or useful system information, can be found. The "cscope" source browsing tool is introduced, and examples are given.

All source files are kept on the "bonnie.engr.sgi.com" system, in the "/proj" directory, which is divided into subdirectories by release name. Underneath the "...release_name/isms" subdirectory, most of the major operating system release components can be found.

File names that end in ".s" are written in assembly language. Most of these are in the ".../isms/irix/kern/ml" subdirectory. File names that end in ".c" are written in the C programming language. File names that end in ".h" are "header files", and are also written in "C".

4-11

22jul1998

TR-IKI rev 0.7b SGI Proprietary

The Difference Between ".h" and ".c" Files

The difference between ".h" and ".c" files is as follows.

Files that end in the characters ".h" are called "header files". Header files can contain either C language source code, or structures and constants used by "modules" (source code files).

Files that end in the characters ".c" are called "modules" or "source files". These files contain "C" language code which, among other things, can include directives to the compiler's pre-processor. Such directives can instruct the compiler to define a name to have a certain value, or to include a certain "header file" as part of the source when the source is compiled.

In a ".c" file, directives that begin with "#include" specify a header file to include in the source. Header file names that are surrounded by the characters "<>", such as:

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

...indicate that certain "standard directories" (this is system and implementation dependent) should be searched and used as a prefix to the file pathname specified. One of the most common of the standard directories is "/usr/include".

```
# cd /usr/include/sys
# ls types.h
types.h
```

Header file names that are enclosed in quotation marks, such as:

```
#include "region.h"
```

...indicate that the directory the source code resides in should be searched first, and, if the file is not found, then the standard directories are searched.

Where to Find ".h" Files

Since header files contain and define the format for kernel structures and pointers, they are very useful to examine, in order to understand kernel functions and solve system dumps.

Not all of the ".h" files will be on your system if you are not running source. For example, "region.h" is not a standard UNIX header file, but is specific to the IRIX methodology of memory management (paging). If you are running a binary version of the operating system at your site, you do not have all the files necessary to compile the operating system on your machine.

If you want to find a specific ".h" file, the best tool to find a source version of it (inside of the SGI firewall) is probably "cscope".

Header (".h") files may also include other header (".h") files. This may make examination of a source file confusing. It may be difficult to understand what structure definitions the code is referencing when the files which define them are not explicitly "included" up at the top of the file. For example, in the directory:

```
/hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix/kern/os/as
```

...one can find the source file "region.c", and the header file "region.h".

The "region.c" file *uses* but does not *include* the "region.h" file. Instead, the "region.c" source file includes a different header file, "as_private.h", which *itself* includes the "region.h" file.

In this way, the region.h header file is included *indirectly* as part of a *different* header file referenced in "region.c". Once inside the SGI firewall, you can use the "cscope" tool to find such occurrences.

In this case, we know (or suspect) that the "region.c" file uses, but does not include, the "region.h" header file. What we need to do is:

- Look at the list of all the different ".h" files that the "region.c" code includes. This can be done by examining the region.c source code directly.
- Make a list of all the files that explicitly include the "region.h" header file. This is one of the standard operations that the cscope tool will do.
- Compare the two lists, that is, compare the "region.c" header files to the list of all the files which include the "region.h" file directly.

and:

- Hope we find a match.

Unfortunately, the last two steps have to be done by hand.

If we find such a file, then we will know that "region.c" *does* include "region.h", indirectly, by way of an intermediate header file which includes "region.h" directly in its directives.

Below is an example of using cscope to try and find which ".h" header files include the "region.h" header file. Then a comparison is done to see if any of these are files used in "region.c".

4-13.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

Files #including this file: /region.h
-----
File                               Line
1  /kern/include/privat             33 #include "region.h"
2  /kern/os/addrmap.c               35 #include "region.h"
3  /kern/os/addrpac.c               39 #include "region.h"
4  /kern/os/addrpac_addrp           34 #include "region.h"
5  /kern/os/addrpac_3elip           35 #include "region.h"
6  /kern/os/addrpac_fault           37 #include "region.h"
7  /kern/os/addrpac_inow            35 #include "region.h"
8  /kern/os/addrpac_lock            36 #include "region.h"

Lines 1-9 of 118, press the space bar to display next lines
Find this symbol:
Find this definition:
Find functions called by this function:
Find functions calling this function:
Find assignments to:
Change this cscope pattern:
Find this cscope pattern:
Find this file:
Find files #including this file: region.h

```

Once you've invoked cscope, a window much like the above is generated. The last command choice option is to do exactly what we want, find all the files which have a directive to explicitly include the "region.h" file. Your keyboard arrow keys will allow you to select which of the nine possible searches you want to invoke.

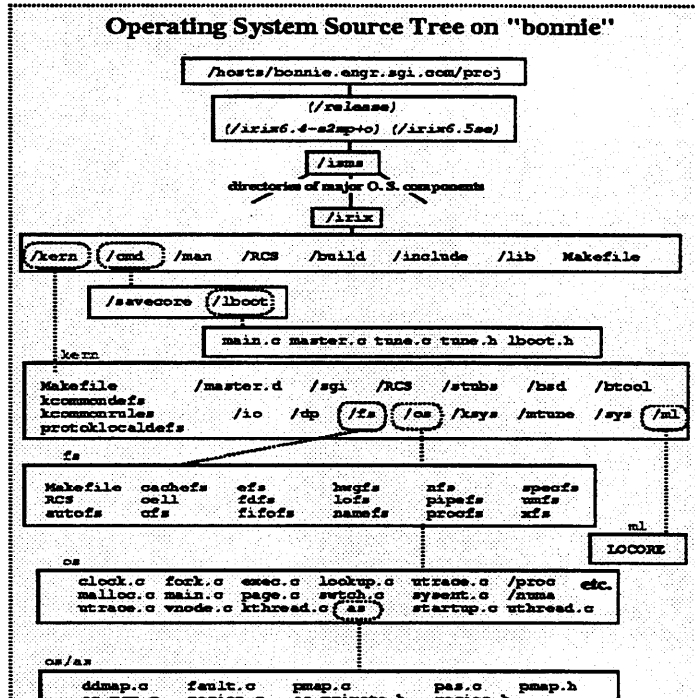
In Step (1), the header file we're interested in, is entered on the appropriate command line. This results in the generation of a list of all files (not just header files) which include the target file. The yellow horizontal bar shows that, in this case, 118 files were found. Pressing your space bar will cycle you through these choices. Many of the result files are source code file names ending in ".c", but we are trying to find a header file (a file ending in ".h"). The area where the file names are listed, however, is not always wide enough for the full file name to show, so you might have to select a file just

4-13.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

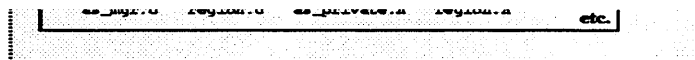
Operating System and Kernel Source Tree File and Directory Structure



4-14

22jul1998

TR-IKI rev 0.7b SGI Proprietary



Once you have gotten onto the "bonnie" system, and selected the operating system release you are interested in from the "/proj" directory, and further descended into the ".../isms/irix" subdirectory, you will find that the "/irix" directory contains the following subdirectories:

```
%cd /hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix;ls
Makefile  RCS      build   cmd     include kern   lib     man
```

The "cmd" subdirectory contains the "savecore" subdirectory, where information about past system crashes or hangs is kept. In the "lboot" subdirectory there are a number of files important to the booting, configuration, and tuning of the system.

```
% cd /hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix/cmd;ls
Makefile  cpr      hinv    mmscd   sn0log   xbstat
RCS       diskless icrash  netman  sn0msc   xfs
bsd       dlpi     ip26ecc nvlog    snmp     xfsm
btool     dprof   lboot   onlinediag stress   xlv
cached    flash   linkstat perfex  systlrd  xperform
clsh      flashio mkmachfile savecore tk
cms       flashmmsc mkpart  smt     tokenring
```

On a live system, much of the configuration and tuning information can be found in the "/var/sysgen" directory. The "/var/sysgen/master.c" file is generated by the boot process and is full of SYSTUNE, system, and driver configuration information. The "bdevsw" and "cdevsw" structures (block special device and character special device switch tables) are also defined in the master.c file.

Most of the kernel code and structure definitions live in the subdirectories under ".../isms/irix/kern".

4-14.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary


```
$ cd /hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix/kern;ls
```

```
Makefile  fs          master.d    sgi
RCS       io          ml          stubs
bsd       kcommondefs mtune       sys
btool    kcommonrules os
dp       ksys      protoklocaldefs
```

Below is an explanation of the subdirectories found in the kernel source tree.

bsd - "bsd" stands for "Berkely Systems Development". Directory contains networking related code, e.g. sockets, protocols, network device drivers.

btool - Contains code for "btool", a code coverage analysis tool.

dp - Contains distributed processing support, ie., cellular IRIX support.

fs - Contains all the code for each of the file systems types, in subdirectories like nfs, pipefs, procfs, xfs, cachefs.

io - Contains source code for I/O device drivers and I/O support routines, e.g., ql.c (SCSI chip logic and qlogics controller code), scsi.c (generic SCSI code), and dksc.c (generic disk drivers).

kcommondefs - Contains basic common flags/locations for kernel builds. The makefile includes these extensions for Make.

kcommonrules - Ensures that kernel builds end up in proper directories, install proper header files, etc. The makefile includes these extensions for Make.

ksys - Contains a directory of kernel private header files - never exported outside the kernel.

master.d - Contains a directory of configuration files for every device driver, which a program called "lboot" reads and decides whether to configure that device drives in or not. If you're writing a new driver, you'd add a new file for it in master.d.

ml - Contains low level machine level code for system startup, locks, interrupt management, and error handling. All the assembler files go here, e.g., MIPS assembly language files, the assembler language level locking code "llsclocks.s", etc.. The ml directory has several interesting subdirectories, including:

LOCORE - A directory of the most common ".s", shared assembler, files for all platforms.

mtune - Contains files with system tunable parameters. These are modified only indirectly using the "systune" tool.

os - A directory containing the bulk of the operating system code, e.g., fork, exec, the main kernel files and directories, the vm directory for virtual memory, the as directory for address space management (part of memory management - NUMA [nonuniform memory access] support code files are in this subdirectory). There is an important subdirectory named "as", which contains most of the code important for page fault handling, the definitions of the region and pregion structures, etc.

protoklocaldefs - The prototype kernel local definitions file. Before a kernel is built, this file is copied into klocaldefs and modified. Binary sites do not need to think about this file.

sgi - This is a directory of somewhat odd codes like a random number generator for the kernel, and some with more obvious value, such as the code for kernel mallocs, kern_heap.c, or chunkio.c, which deals with DMA (Direct Memory Access - I/O controllers get data out of memory by touching it directly, they don't go through the CPU. The "chunkio" code coalesces DMAs to do large efficient direct memory accesses instead of lots of small inefficient ones).

stubs - If there's an optional subsystem, that some site doesn't want, a stub for it is put in here and when that code gets referenced the code will just return.

sys - This is where *most* of the public header files, i.e., those that get installed in /usr/include/sys can be found. But some of the public header files are in /irix/kern directories and get exported. This is also true of some of the private header files. Many kernel structure definitions can be found in this directory

NOTE: Graphics has commands and libraries and kernel pieces, but it's not in /irix/kern. However, many related pieces, such as header files and device drivers, can be found under other "isms", for example, isms/gfx/kern, and the digital media drivers in /isms/dmedia/kern. The communications group has its own kern header files, such as the IBM x25 communications support under /isms/comm/kern.

NOTE: The IRIX 6.4 source code is scattered all over, but the "isms" directory should contain the complete list of independent software modules. There are exceptions to this, however. For example, the compilers group has its own release schedule, since compilers are unbundled. Compiler versions can be found in subdirectories under:
`/hosts/bonnie.engr.sgi.com/isms/cmplrs.src`

Tools Available to Browse Source

- cscope - interactively examine a C program
 - Tutorial available
 - See man page on "tokyo" (login as guest)
 - Not officially supported, just for internal use
 - Supported equivalent is "gid"
 - see man page on Indys
- dwarfdump - locate source patches
 - Example available
- ctags - create a tags file
 - For vi users
 - See man page
- etags - create a tags file
 - For emacs users
 - No man page

(note: there is an Eagan cscope database of released IRIX 6.4 source)

(note: there is an Eagan cscope database of patched IRIX 6.4 source)

(note: there is an Eagan cscope database of development IRIX 6.5 source,

Determining What Software the System Is Running

- versions
 - Show system software
 - List installed patches
 - "versions -Inv | grep patch"
 - "versions -bv | grep patch"
 - Remove system software (eg, patches)
 - Has man page
- uname
 - Show system software
 - "uname -a"
 - "uname -R"
 - Has man page

versions - show system software; list installed patches

See "man versions".

Versions has many options and three main functions.

The "showprods" option displays information about the software that is currently installed on a system.

"Showfiles" displays lists of files on your system and information about those files ("inst" can be used to remove installed software from your system).

Typing "versions -Inv | grep patch", or "versions -bv", will generate a list of patches installed on the system (versions can also be used to remove patches).

How Do I Know What Crashed My System?

What Was Going On Just Before the System Crashed?

- icrash- IRIX system crash analysis utility
 - Has man page
 - Tutorials available
 - Tips available
 - Gives status of
 - networks
 - disks
 - tapes
 - OS state
 - PE states
 - register contents
- SYSLOG - writes message onto the system log
 - Has man page
- utrace - Basic kernel trace mechanism (NEW 12/97)
 - Has web page
 - Circular buffer of time-stamped events for each CPU
 - Requires kernel rebuild to enable

Where Does the System Put Things When It Crashes?

System logs are in either:

- */usr/adm*
- or
- */var/adm*

What System Logs Exist?

System Logs in */var/adm*:

```
$ cd /var/adm ; ls
```

```
SYSLOG      dtmp      pacct1    pacct4    pcplog    utmp
acct        fee       pacct10   pacct5    sa         utmpx
avail      klogpp   pacct11   pacct6    sat        wtmp
bds.log    lastlog  pacct12   pacct7    sulog     wtmpx
crash      mkpts    pacct2    pacct8    sysmon.msg
dodiskerr  pacct    pacct3    pacct9    sysmonpp
```

4-23

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Description of system logs, subdirectories, and files:

Most of the interesting system logs, subdirectories, and files are in */var/adm* .

SYSLOG - log of everything happening on the system. See man SYSLOG.

acct - invoke accounting. See man acct.

avail - directory of logs , see */var/adm/avail/availlog* - the primary log of the availmon tool, which keeps track of when you bring the machine up and down and why, and, if so configured, will send mail to SGI headquarters. The log contains information that this machine was rebooted, and for what reason, how long was it down etc. That log is processed and there's a database that tries to summarize field info.

bds.log - logs all the opens and closes of and performance data for BDS (Bulk Data Services files (which are used to transfer large quantities of data between machines).

crash - where the dumps are put (default directory) .

dodiskerr - part of disk error accounting

dtmp - output from the acctdusg program

fee - output from the chargefee program, ASCII tacct records

klogpp - symbolic link to */usr/sbin/klogpp*, which is the command that filters kernel messages for the syslog (to SYSLOG and the */dev/console*)

lastlog - record of who's logged in.

pacct - raw data file of all accounting activity that the acct command reads into reports

4-24

22jul1998

TR-IKI rev 0.7b SGI Proprietary

pcplog - performance copilot log

sa - unix acct

sat - directory of security audit trail related files

sulog - record of who tried to su to root or to some other login ID

sysmon.msg - sysmon allows a user to browse SYSLOG; this file contains a message from the SYSLOG file.

sysmonpp - program filter for log messages

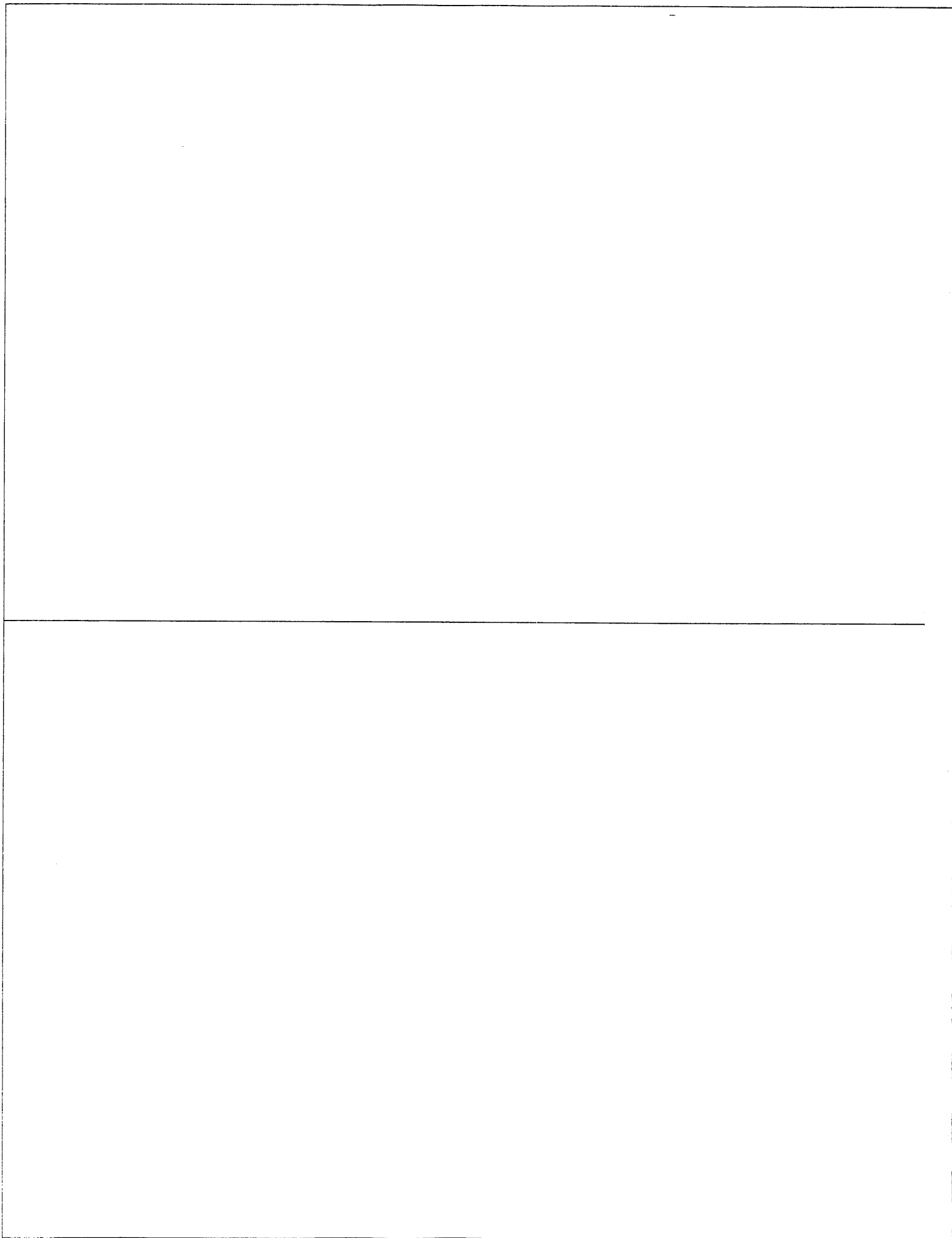
utmp - see man page man pages for these last 4. These are logs of who logged in to do what commands on which terminal. These files hold user and accounting information for such commands as who, last, write, and login.

utmpx - see "utmp", above.

wtmp- see "utmp",above.

wtmpx- see "utmp",above.

A more involved exploration of code examination and system dump analysis is handled in other modules.



Module 5: Operating System Overview

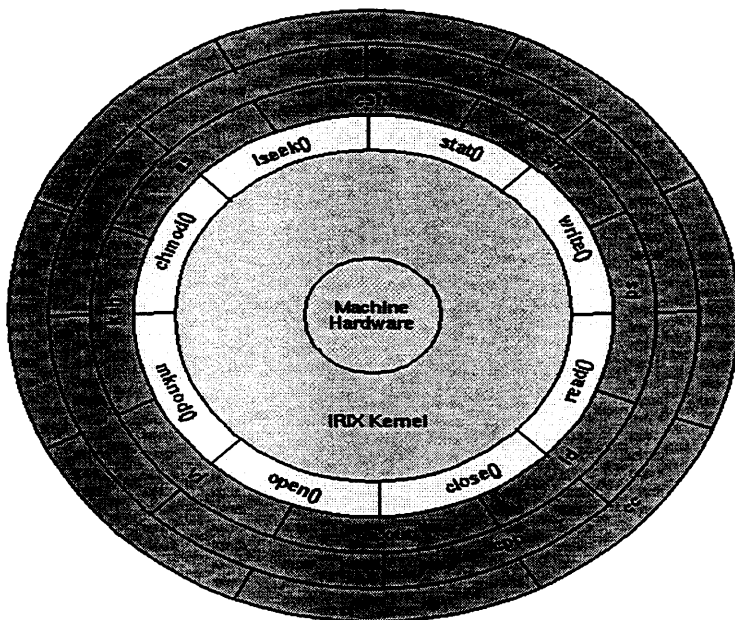
IRIX Operating System Overview

This section provides an overview of the organization of the IRIX operating system, user memory components, memory management, process relationships, and the primary functions of the IRIX kernel code.

By the end of this section, the student should be able to:

- Explain the IRIX operating system philosophy
- Explain the concept of an interrupt
- Explain the concept of an exception
- Describe the major system components of system memory
- Describe the major system components of user memory
- Explain how kernel and user components are related
- Explain the primary memory management methodologies for moving pages in and out of memory
- Describe the relationship of `sched` and `init` to all other processes in the system
- Describe the functions of the `fork` and `exec` system calls
- Describe the process relationships for a user connecting to a system through a network
- Explain each of these primary kernel activities:
 - System Initialization
 - Process Management
 - User Program Interface
 - Memory Management
 - File Management
 - I/O Management
 - Communication Facilities

UNIX (IRIX) philosophy

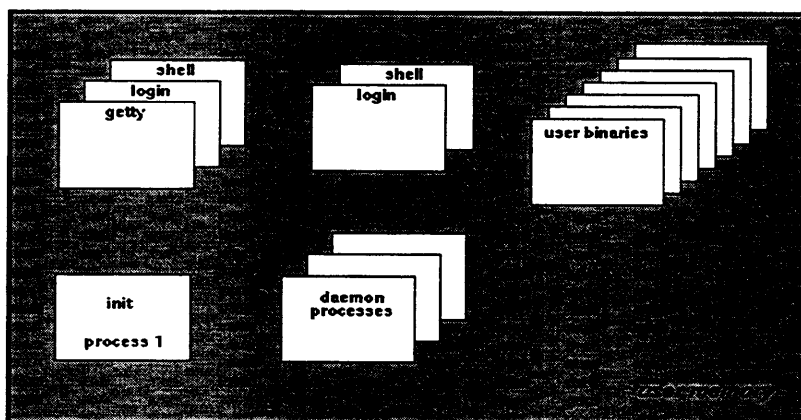


The philosophy of UNIX (IRIX) is to take advantage of work already done by others. As this onion-like diagram

suggests, UNIX (IRIX) is built in layers, with each layer representing a building block that can be used to build other building blocks. Most commands, programs, and utilities supplied with the UNIX (IRIX) system can be used in combination with each other to build other tools. Complex mechanisms can be built from a set of simple commands to perform various functions.

The UNIX (IRIX) operating system kernel, in most instances, insulates the user from needing to know intimate details of the machine hardware. The machine hardware would include processors, peripheral devices, memory, hard disks, etc. The layer immediately outside the IRIX kernel is referred to as the *system call* layer. System calls allow user-written applications at the outer layers to invoke various functions residing inside the IRIX kernel. For example, an application needing to read a file would issue the `read()` system call which would invoke a *read* handler inside the kernel that would communicate with the device where the desired data resides and return the data to the application.

IRIX system major components (user memory)



The above diagram illustrates the major components that comprise user memory in an IRIX system.

The `init` process (`pid(1)`) is created and started by the first kernel process, `sched` (`pid(0)`) during system start-up and becomes the parent of all other processes (except `sched`) in the system. The `getty`, `login`, and `shell` (`sh`, `csh`, `ksh` ...) processes provide the interfaces between the system and users.

Several daemon processes exist to provide services to both users and the kernel. The network daemons (`inetd`, `telnetd`, etc.) provide the interface between the system and a user at a network terminal. Daemons such as `nsd` and `cron` execute user scripts or commands non-interactively. Other daemons provide functionality for the kernel. The kernel can "off-load" lengthy work to them, such as tape support, accounting, error logging, and so on.

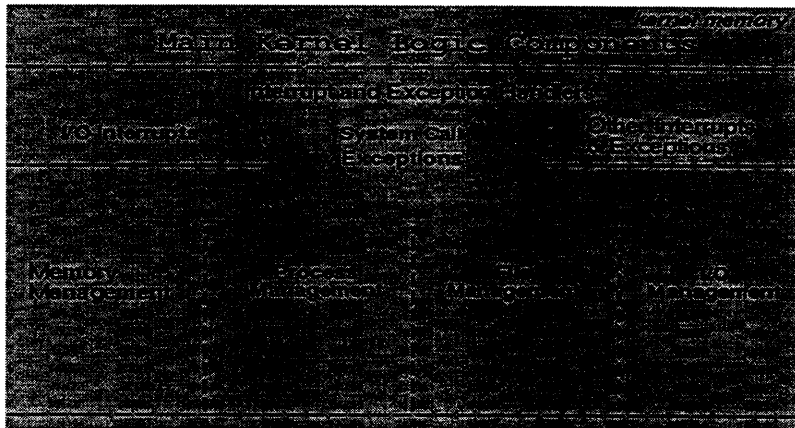
User binaries represent the execution of program binaries as initiated by the shells.

5-3.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

IRIX system major components (kernel logic)



This diagram illustrates the major components that comprise the IRIX operating system kernel.

The kernel logic is the overall controlling component in the system.

5-4

22jul1998

TR-IKI rev 0.7b SGI Proprietary

When Does the Kernel Take Control Away From a User Process?

Kernel code will take control of a CPU away from a user process when:

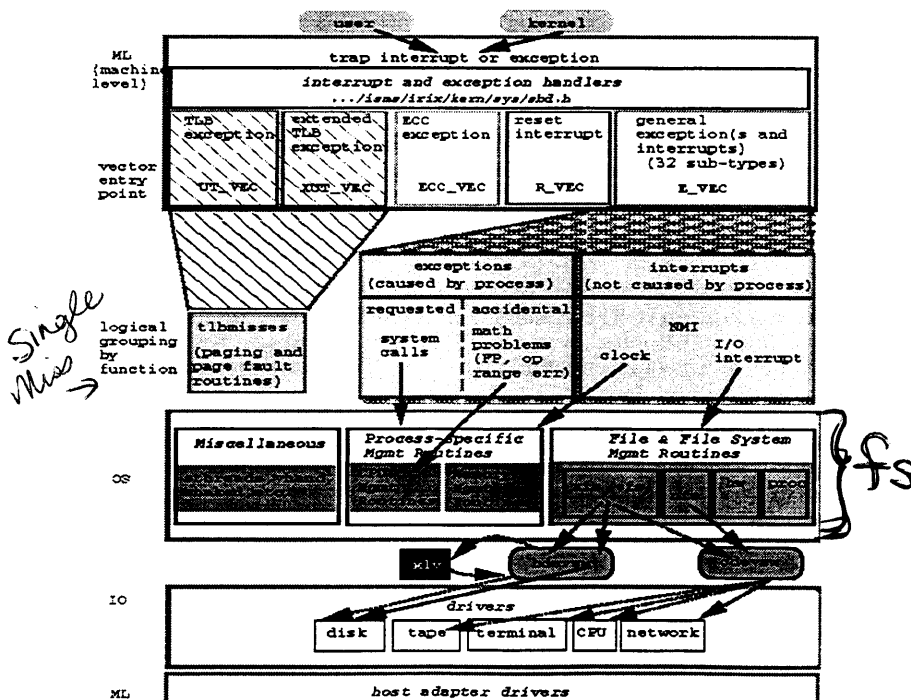
- The user process receives an *interrupt*

An interrupt is something generated externally to the process, which requires kernel intervention, such as the completion of I/O.

- The user process generates an *exception*

An exception is something generated by the process, such as when a user process makes a request to the kernel to take control of the CPU to do a system call.

Kernel block diagram



{machine
level}

SCSI disk

SCSI tape

VME

5-6.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Primary Kernel Activities

Kernel code selects one of several handlers to service the interrupt or exception. After all interrupts and the user's exception (if any) have been processed, the kernel will return control of the CPU to a user - but it may not be the same user who had the CPU before the kernel call. Which user the kernel connects to is determined by which process has the highest priority to run at that time.

A major part of the kernel code provides for the processing of system calls. A *system call* is a type of interruption to user processing, called an "exception". System calls can be organized into one of four areas: process management, file management, I/O management, and miscellaneous routines that are used by both the kernel and "outside" processes.

Another major portion of kernel logic is devoted to handling I/O interrupts, which signal an I/O completion.

On IRIX systems, the primary memory management methodology is based on moving *pages* in and out of memory, not *processes*, so invoking `sched` for that purpose is done as a last resort. The primary memory management routines for moving pages *into memory* are kernel routines triggered by a "TLB miss", when a CPU cannot find what it wants in its Translation Lookaside Buffer, and must ask the kernel to make page information more local. The primary memory management routines for moving pages *out of memory* start with `vhand`.

To facilitate performance, a copy of the IRIX kernel resides in that part of main memory assigned to each individual node of a multi-node system. Interrupts and exceptions generated by user processes, interprocess communication, system calls, etc. can be handled more efficiently (faster) when a copy of the IRIX kernel is located "nearby" in the node's local memory instead of a CPU having to access kernel code through the interconnect fabric from a part of memory that would not be considered local, and would therefore take more time to access.

5-7

22jul1998

TR-IKI rev 0.7b SGI Proprietary

The kernel block diagram above shows various user and kernel modules and how they are related. This is a useful model, although interactions in the kernel are much more complex than this.

The UNIX (IRIX) kernel is designed around two primary entities: files and processes. Therefore, two major components of the kernel are the file subsystem and the process control subsystem.

The block diagram shows three levels: user, kernel, and hardware. The system call interface represents the boundary between user programs and the kernel. A *system call* is a request made by a user's program to execute a function residing in the operating system kernel. Library functions, which also invoke system calls, are actually linked together with the user's program.

The diagram partitions the set of system calls into two groups; those that interact with the file subsystem and those that interact with the process control subsystem. The file subsystem manages the creation and removal of files, controls access to files, allocates file space, administers free space, and reads and writes data for users. A user's process interacts with the file subsystem using system calls like `open(2)`, `close(2)`, `read(2)`, `write(2)`, `chmod(2)`, `chown(2)`, and `stat(2)`.

The file subsystem provides user access to data using a buffering mechanism that controls the flow of data between the kernel and secondary storage. The kernel's buffering mechanism interacts with block I/O device drivers to initiate reads and writes of data to and from the kernel. Device drivers are kernel modules which control access to peripheral devices. A block I/O type of device is a device which is read and written in fixed units, referred to as a block, or multiples of a block. Data residing on a block I/O devices can be accessed in a random manner. An example of block devices would include disk drives.

The file subsystem also interacts with character devices. Character devices include all devices which are not block devices, and can be read and written to by as little as one character at a time, such as terminals and tape devices.

The process control subsystem has responsibility for the creation/termination of user processes, interprocess communication (IPC), process scheduling, process synchronization, and memory management. A user's process interacts with the process control subsystem using system calls like `fork(2)`, `exec(2)`, `wait(2)`, `exit(2)`, `brk(2)`, `kill(2)`, and `signal(2)`.

The memory management facility is responsible for making sure each process is allocated sufficient memory to perform its tasks. IRIX uses *demand paging* to control user memory space. "Demand paging" means that a page containing the requested information is made local to a CPU only when it is needed, or "demanded", by the executing process.

The scheduler facility is responsible for fairly allocating the CPU's to individual user processes. This is handled through queuing mechanisms. Processes with the highest priority are given CPU attention first. A process either voluntarily gives up its CPU while waiting for a resource (for example, I/O data or system call handling) or the process is preempted by the kernel when its time slice is consumed.

Interprocess communication is supported in several forms including signals, pipes, shared memory, and message queues.

Hardware control is responsible for handling device interrupts. Devices like terminals or disks may interrupt the CPU while a process is executing. Interrupts are serviced by special interrupt handling functions in the kernel.

Summary of IRIX Kernel Primary Functions

- **System Initialization**

A facility exists for the IRIX kernel to start up and initialize itself. The system provides a "bootstrap" facility to load a copy of the IRIX kernel into the system memory and start running.

- **Process Management**

A facility to create, terminate, and control user processes. IRIX is a multiprocessing operating system, so the kernel ensures that each active user process is given its appropriate share of CPU attention and other resources. Therefore, all processes appear to execute in parallel.

- **User Program Interface**

The kernel provides a robust set of system calls allowing user programs to access the vast array of services provided by the operating system. System calls are invoked by library routine interfaces to the operating system.

- **Memory Management**

On IRIX systems, the total amount of memory needed to accommodate all currently active processes far exceeds the physical memory installed in the hardware. To simulate more memory than is physically available and help overcome this bottleneck, the IRIX kernel implements a virtual-memory system. The system maps virtual addresses to physical addresses at run time. Therefore, there are no memory restrictions on a user's process other than those imposed by the operating system or imposed by the system administrator.

- **File Management**

The IRIX operating system maintains many types of files which reside in file systems. A file system is an organized hierarchy of directories containing these various file types. File systems typically reside on physical media such as hard drives and the operating system provides the services to access individual files within file systems. IRIX supports multiple file system types.

- **I/O Management**

The operating system provides several user-selected options to influence the path taken for input/output data, which affects I/O performance and the level of risk for data loss. The kernel supports familiar I/O methods such as sequential and random I/O, buffered and direct I/O, synchronous and asynchronous I/O, file locking mechanisms, etc.

- **Communication Facilities**

The operating system provides for inter-process communication, inter-machine communication (networks), and communication between processes and devices.

Module 6: Interrupt and Exceptions (Preliminary)

Interrupts and Exceptions (Preliminary Notes)

This section provides an overview of Interrupts and Exceptions. By the end of this section, the student should be able to:

- describe the difference between an interrupt and an exception
- describe the five different initial entry points, or "vectors", into the kernel for interrupts and exceptions
- describe the logic flow through various interrupt and exception handlers

Processor Operating Modes

The MIPS processor under IRIX operates in one of two modes: kernel and user. The processor enters the more privileged kernel mode when an interrupt, a system instruction, or an exception occurs. It returns to user mode only with a "Return from Exception" instruction.

Certain instructions cannot be executed in user mode. Certain segments of memory can be accessed only in kernel mode, and other segments only in user mode.

Interrupt and Exception Types

- **Types and Entry Points (handler table)**
 - **TLB exception**
 - **Extended TLB exception**
 - **ECC Exception**
 - **Reset Interrupt**
 - **General Exception(s and Interrupts)**
 - **32 subtypes**

The hardware defines the entry points. The handling is done by software.

The processor supports five hardware, two software, one timer, and one nonmaskable interrupt. The hardware Interrupt is described in great detail in Chapter 17 (17.3) of the R10000 Microprocessor User's Manual, in the section titled "Interrupt Exception" (http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/HTML/t5.Ver.2.0.book_365.html#0).

Software exceptions and interrupts are described in great detail in Chapter 6 (6.14) of the R10000 chip manual, in the section titled "Interrupts" (http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/HTML/t5.Ver.2.0.book_129.html#HEADING169).

How are Interrupts Different From Exceptions?

- **Interrupt**
 - **asynchronous to the currently executing process or thread**
 - **due to causes unrelated to the current user process**
 - **After an interrupt, control returns to the next instruction**
- **Exception**
 - **synchronous to the currently executing process or thread**
 - **caused by, or requested by, the currently executing process or thread**
 - **After an exception, control returns to the same instruction**

Exceptions are occurrences which make a CPU stop operating in user mode and begin executing in kernel mode, as a direct result of something that user's process did, either accidentally (such as a floating point error), or on purpose (such as a system call request).

Some examples of exceptions are floating point exceptions, system call exceptions, page fault exceptions.

Interrupts are (probably) not due to actions of the currently connected process. If a CPU receives an I/O interrupt, the CPU will stop executing in user context, and start executing in kernel context, in order to handle the I/O interrupt. The I/O which has completed is probably the last steps of some other process's I/O system call request, although it could be an asynchronous I/O which was requested earlier by the currently connected process. An interrupt is externally, asynchronously, caused, and distinct from the currently executing process.

Some examples of interrupts are disk interrupts, tty interrupts, hardware error interrupts, clock interrupts.

Another difference between exceptions and interrupts is where the CPU resumes execution in the user process, once the

CPU returns to user context. In general, exceptions occur "in the middle of instructions", therefore when the CPU returns to user context, it has to try to restart that same original instruction. And interrupt occurs "between" instructions, that is, when the CPU returns to user context, it executes the *next* instruction after the point the interrupt occurred.

6-4.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

How are Interrupts Similar to Exceptions?

- Both cause the CPU to do an exchange to kernel mode.
- Kernel then saves context of previous process

6-5

22jul1998

TR-IKI rev 0.7b SGI Proprietary

MIPS Processor Exception and Interrupt Kernel Entry Points

On MIPS processors, there are five possible interrupts or exceptions, but only FOUR entry points to the kernel:

1. TLB exception
2. Extended TLB exception
3. ECC exception
4. Reset interrupt
5. General exception

The Reset Interrupt, like the NMI Interrupt, is caused by pushing a button, and is handled by the hardware, not by the kernel.

Examine the "sbd.h" file (use cscope, or look on look on bonnie for the source file. Here's the path on bonnie for the IRIX 6.5 version /hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix/kern/sys/sbd.h).

(I think "sbd" stands for "system board" ?)

This file defines the five entry points listed above. Although the comments say "Chip definitions for R3000 and R4000", these entry points apply to the R10000 chip as well.

All of them are called "Exception vectors" (see at or about line 39).

The TLB Exception

The first one, the "utlbmiss vector", listed above as the "TLB exception", is defined at (or about) line 49.

```
#define UT_VEC          COMPAT_K0BASE          /* utlbmiss vector */
```

The Translation Lookaside Buffer is a piece of hardware used to contain mappings of virtual addresses to physical addresses. It is of limited size. With an R10000 chip, there are 64 registers, each of which holds two virtual-to-physical page mappings, so there is a maximum of 128 possible translations from virtual to physical that can be found in the TLB at any given time. With an R4000 or R5000 chip, there are 64 registers, each holding only one virtual-to-physical mapping. If a user tries to reference a virtual address that isn't one of the current 64 or 128 in the hardware, we take a "TLB exception" or "TLB miss" (these terms are synonymous), and the MIPS processor then starts executing code at the entry point for TLB exceptions (at address 80000000).

The Extended TLB Exception

```
#define XUT_VEC          (COMPAT_K0BASE+0x80)  /* extended address tlbmiss */
```

This is the entry point for the "extended TLB handler". This handler has the same function as the TLB miss exception vector, but handles TLB misses on 64 bit addresses. The utlbmiss vector handles TLB misses on 32 bit addresses, which is the default.

TLB misses don't have time to do anything but that, so the code doesn't handle anything much more than the TLB miss situation.

The ECC Exception

```
#define ECC_VEC          (COMPAT_K0BASE+0x100) /* Ecc exception vector */
```

ECC exception code (Error Correcting Code exceptions) is used to handle single or multi-bit errors, or single or multi-bit cache errors. All MIPS processors jump to the ECC exception vector, at yet another "well-known place in memory". The R10000 puts this entry point at a fixed place in memory. It's unusual that an ECC exception happens, and it doesn't

follow normal rules for handling. In general, the more memory your system has, the more frequently you will get ECC exceptions. You probably won't see these very often and it will be obvious when you do. The machine software and hardware will call out where the problem area is (if it's a double-bit error) and keep track of how often and if it is going bad (single-bit errors).

Like the TLB exception handlers, ECC exceptions handle the situation and don't do much of anything else. Both the TLB handlers and the ECC handler are explicitly short exceptions.

The Reset Interrupt

```
#define R_VEC          (COMPAT_K1BASE+0x1fc00000)    /* reset vector */
```

The non-maskable interrupt and the reset interrupt and the power on interrupt are all caused by people pushing buttons. When a processor receives an NMI (Non-maskable interrupt), or reset interrupt or power on interrupt, it doesn't actually get handled by the kernel. As a result of the button being pushed, the CPU goes off into PROM to handle it, so these are really not handled by the kernel.

Nonmaskable Interrupt (NMI) Core Dumps

It is possible to manually generate a system core dump without the benefit of a system panic. All high-end (CHALLENGE® L or CHALLENGE® XL servers, Onyx® workstations, Origin200 and Origin2000 systems) systems contain a special feature that enables system administrators to initiate system core dumps. They accomplish this by issuing a Nonmaskable Interrupt (NMI) request. Depending on the system, selecting a system controller menu option or pressing a special button on the system controller will initiate an NMI. A system administrator (often at the request of SGI support) normally induces an NMI core dump when users of a system complain that the system is partially or completely hung. The resulting system core dump may provide a clue about the cause of the problem.

The General Exception(s and Interrupt vector)

6-6.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
#define E_VEC          (COMPAT_K0BASE+0x180)    /* Gen. exception vector */
```

Calling this vector the "General *exception* vector" is a bit of a misnomer. A better name would be the "General Exception AND INTERRUPT vector", because it actually is for both interrupts and exceptions. This entry point handles all the kinds of interrupts and exceptions not handled by the other vectors (eg., system calls, clock interrupts, I/O, etc.).

On the MIPS architecture, all the reasons you'd end up here are subdivided into 32 different ways of dealing with why-you-got-here, sort of like 32 sub-entry-points, 32 potential vectors within the general exception vector, and those 32 are found in

```
.../irix/kern/os/startup.c
```

Look for "causevec".

There's a vecint vector for interrupts, more TLB stuff, read misses, write misses, attempts to modify stuff when you don't have permission, read/write address errors, system calls, breakpt instructions, etc. FP overflow see MIPS architecture manual for a definition of each of these all MIPS architecture manual: in SGI home page. See:

http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/HTML/t5.Ver.2.0.book_1.html

In particular, go the table of contents and take a look at Chapter 17 on CPU Exceptions and 6.14 on Interrupts. In that same book, go to the index, find "Cause register".

The MIPS architecture defines 8 interrupt levels defined in the description of the coprocessor 0 status register and the coprocessor 0 cause register

Some background:

MIPS defines things as part of the CPU. One of the concepts of "CPU" is the concept of 'coprocessor'. there used to actually be a co-processor, but now it's actually built into the CPU chip - but it's still called "the co-processor". In all

6-6.c

22jul1998

TR-IKI rev 0.7b SGI Proprietary

MIPS cpus, coprocessor 0 is for the special CPU control registers, manipulate TLB, status regs, cause regs, clock reg, count and compare regs, up to about 32 control regs - see the architecture manual. These can only be accessed in kernel mode. Coprocessor 1 is always the Floating point control registers processor, so anytime you have any FP status reg and FP control reg, for FP operations. The architecture also defines Coprocessors 2 and 3, but they are not yet in use (possibly will be for media extensions and vector extensions).

Actually, these are register sets, pieces of the CPU. The word "processor" is misleading. Every CPU has these.

The status register in coprocessor 0 has 8 hardware levels defined for interrupts and we don't use them. Because that's not enough on the big machines. we have so many - so on Origin and other platforms, we provide, external to the CPU hardware, in the HUB chip, a register that says *here* are the interrupts that are really pending and they are prioritized into 128 levels, pretty much per-device, eg, for each SCSI controller, it's common to use 30-40 of them, as defined by MV developers. The hardware prioritizes them but the software just uses them as 128 levels. same as UNICOS concept of hardware prioritizes by bit position versus software doesn't care re/interrupts. The HUB register is called the interrupt pending register.

To see all about how to read what interrupt is pending in the register, see:

<http://babylon.engr.sgi.com/systemsw/projects/lego/hardware.html>

(SNO used to be called Lego)

click on Lego Hub, Router, IOC3, LINC Chips
ChipDoc - Chip
Hub Programming Manual
chapter 2
CHAPTER 2 Hub Internal Register Definitions
2.1.1.20 INT_PEND0
2.1.1.21 INT_PEND1

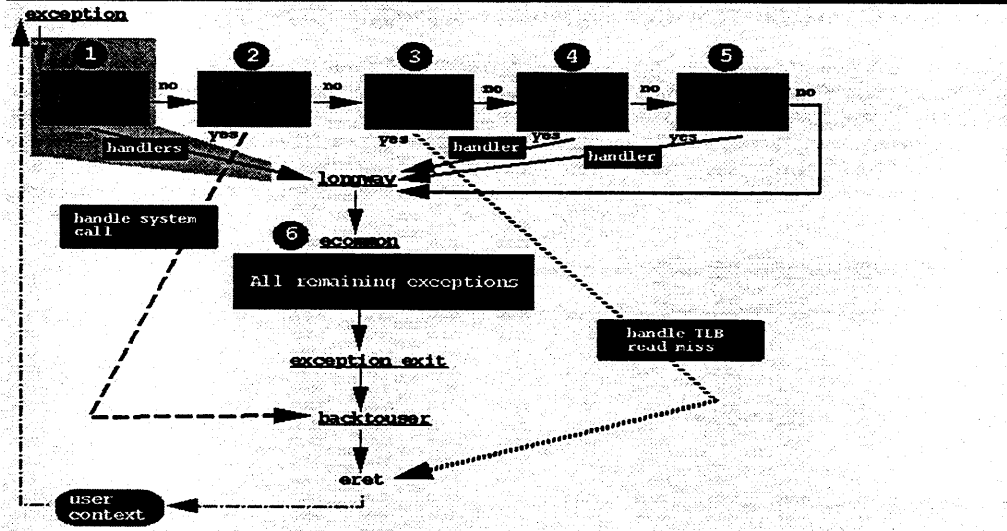
What's important is that there is system hardware external to the processor that keeps track of which interrupts are

actually pending at a given time, but these aren't icrash-locatable, so there's no way to find out what the processor actually knows.

So that's a quick walk-through of the entry points for the four/five exceptions/interrupts mentioned before. When it comes to handling interrupts and exceptions, there are special handlers for each of those 5. Four of them handle special cases, and the fifth one has the 32 subtypes in a table in `startup.c`. If you look at the table, you'll see there's a single subtype for all device interrupts (disk, tape, console, tty, etc.). We always comes through the general exception vector, and always check to see if it's a hardware interrupt. If it is, the CPU spawns a thread for that kind of hardware, and for the device handler, which has other subtypes, and looks at more and more sub-divisions of handlers until it gets to point where some piece of software says *this* is the driver and the interrupt handler for this particular device, and spawns a thread to handle it (thread is spawned higher up).

General Exceptions

General Exception Vector:
 Hardware Interrupt and Hardware/Software Exception Check Sequence
 /hosts/bonnie.engr.sgi.com/proj/release/isms/irix/kern/ml/LOCORE/gen_exc.s



The above illustration is a simple diagram of the kinds of interrupts and exceptions that the kernel checks for, and reflects

6-7

22jul1998

TR-IKI rev 0.7b SGI Proprietary

the checks that the code goes through in the actual order that they are checked for.

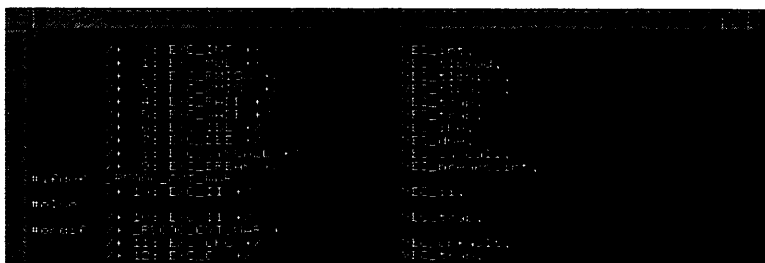
Details of how these checks are performed, and more detail about the different interrupt and exception types is below.

Five interrupt and exception vectors were described earlier (see /hosts/bonnie.engr.sgi.com/proj/release/isms/irix/kern/sys/sbd.h):

```
#define UT_VEC          COMPAT_KOBASE          /* utlbmiss vector */
#define XUT_VEC        (COMPAT_KOBASE+0x80)  /* extended address tlbmiss */
#define ECC_VEC        (COMPAT_KOBASE+0x100) /* Ecc exception vector */
#define R_VEC          (COMPAT_K1BASE+0x1fc00000) /* reset vector */
#define E_VEC          (COMPAT_KOBASE+0x180) /* Gen. exception vector */
```

The "UT_VEC" and "XUT_VEC" exceptions handle 32-bit and 64-bit architecture TLB misses, respectively. The ECC_VEC, or "Error Correcting Code" vector handles single or multi-bit errors, as well as single or multi-bit cache errors. The "R_VEC", or "Reset" vector handles NMI (Non-Maskable Interrupt) and reset interrupts.

The "Interrupt and Exception Roadmap" details the "E_VEC", or "general exception" vector, which actually handles both hardware interrupts as well as software interrupts and exceptions. There are 32 possible types of "general exception", as defined in .../isms/irix/kern/os/startup.c, shown in the cscope screen extract below.



6-7.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary


```

+ 17: E_ILTRAP *      REC_traps
+ 18: undefined *      REC_trap
+ 19: undefined *      REC_trap
+ 20: undefined *      REC_trap
+ 21: undefined *      REC_trap
+ 22: E_ILTRAP *      REC_traps
+ 23: E_ILTRAP *      REC_traps
+ 24: undefined *      REC_trap
+ 25: undefined *      REC_trap
+ 26: undefined *      REC_trap
+ 27: undefined *      REC_trap
+ 28: undefined *      REC_trap
+ 29: undefined *      REC_trap
+ 30: undefined *      REC_trap
+ 31: undefined *      REC_trap
+ 32: E_ILTRAP *      REC_traps
+ 33: undefined *      REC_trap
+ 34: undefined *      REC_trap
+ 35: undefined *      REC_trap
+ 36: undefined *      REC_trap
+ 37: undefined *      REC_trap
+ 38: undefined *      REC_trap
+ 39: undefined *      REC_trap
+ 40: undefined *      REC_trap
+ 41: undefined *      REC_trap
+ 42: undefined *      REC_trap
+ 43: undefined *      REC_trap
+ 44: undefined *      REC_trap
+ 45: undefined *      REC_trap
+ 46: undefined *      REC_trap
+ 47: undefined *      REC_trap
+ 48: undefined *      REC_trap
+ 49: undefined *      REC_trap
+ 50: undefined *      REC_trap
+ 51: undefined *      REC_trap
+ 52: undefined *      REC_trap
+ 53: undefined *      REC_trap
+ 54: undefined *      REC_trap
+ 55: undefined *      REC_trap
+ 56: undefined *      REC_trap
+ 57: undefined *      REC_trap
+ 58: undefined *      REC_trap
+ 59: undefined *      REC_trap
+ 60: undefined *      REC_trap
+ 61: E_ILTRAP *      REC_traps
+ 62: undefined *      REC_trap
+ 63: undefined *      REC_trap
+ 64: undefined *      REC_trap

```

When a CPU acts on an "E_VEC", or general exception, it executes the code found in `.../release/iris/irix/kern/ml/LOCORE/gen_exc.s`. The primary function of the `gen_exc.s` code is to determine which of the above 32 reasons caused the CPU to stop operating in user context, and what is the appropriate routine to handle what must be done while executing in kernel context.

The `gen_exc.s` code checks first for interrupts, then for system calls, by examining the bits in the `k0` register after masking them against `CAUSE_EXCMASK` and then `EXC_SYSCALL`, as shown in the `cscope` screen extract, below.

```

and    k0,LOCL_EXCMASK    # isolate exception cause
and    k0,k0,EXC_SYSCALL # extract

```

After the above, checks are made for a TLB read miss (exception), watch exception, and breakpoint exception, after which the code falls through to a routine which handles "everything else".

The `CAUSE_EXCMASK`, `EXC_SYSCALL`, and other exception comparison bit fields are explained in the following `cscope` screen extracts.

Hardware *Interrupt* Check

The possible hardware interrupts are defined in `.../release/isms/irix/kern/sys/sbd.h` :

```
* Interrupt pending bits
#define CAUSE_IP0 0x00000000 /* E_ternal level 0 pending */
#define CAUSE_IP1 0x00004000 /* E_ternal level 1 pending */
#define CAUSE_IP2 0x00008000 /* E_ternal level 2 pending */
#define CAUSE_IP3 0x0000c000 /* E_ternal level 3 pending */
#define CAUSE_IP4 0x00010000 /* E_ternal level 4 pending */
#define CAUSE_IP5 0x00014000 /* E_ternal level 5 pending */
#define CAUSE_IP6 0x00018000 /* E_ternal level 6 pending */
#define CAUSE_IP7 0x0001c000 /* E_ternal level 7 pending */
#define CAUSE_SW0 0x00020000 /* Software level 0 pending */
#define CAUSE_SW1 0x00024000 /* Software level 1 pending */
```

For an R10000 chip O2000 or O200 machine, the hardware interrupt bit mask is explained in `.../release/isms/irix/kern/sys/SN/SN0/IP27.h` :

```
* R10000 status register interrupt bit mask usage for IP00.
#define SRB_SUITNO CAUSE_SW1 /* 0x00010000 */
#define SRB_NET CAUSE_SW0 /* 0x00000000 */
#define SRB_DEVO CAUSE_IP0 /* 0x00004000 */
#define SRB_DEVA CAUSE_IP1 /* 0x00008000 */
#define SRB_TIMOCLK CAUSE_IP2 /* 0x0000c000 */
#define SRB_PROFCLK CAUSE_IP3 /* 0x00010000 */
#define SRB_ERR CAUSE_IP4 /* 0x00014000 */
#define SRB_SCHEDCLN CAUSE_IP5 /* 0x00018000 */
```

```
#define SRB_ERRNTIMO_IDX 0
#define SRB_ERRNET_IDX 0
#define SRB_ERRDEVO_IDX 0
#define SRB_ERRDEVA_IDX 1
#define SRB_ERRTIMOCLK_IDX 2
#define SRB_ERRPROFCLK_IDX 3
#define SRB_ERRERR_IDX 4
#define SRB_ERRSCHEDCLN_IDX 5
#define NULL_CAUSE_INTRS 6
```

The priority and definition of the above hardware interrupts can be found in `.../release/isms/irix/kern/ml/SN/intr.c` :

```

Priority of GCRDE register interrupts- top priority first.
SRB_ERR      IPR - Hub Errors
SRB_NEWI     IPR - Device and hub errors
SRB_PROFCLK  IPR - Profiling clock
SRB_SCHEMCLK IPR - Scheduling clock

SRB_TIMOCLK  IPR - Realtime clock (RT)

SRB_DEVO     IPR - Devices
SRB_NET      IPR - Net/net
SRB_TIMEOUT  IPI - Soft timeout

```

Software and Hardware Exception Check

The possible hardware and software exceptions, like the hardware interrupts, are also defined in `.../release/isms/irix/kern/sys/sbd.h`. Notice that if the result of the masking operation is a zero, we have a hardware interrupt of one of the kinds defined above. As state above, this is the first thing the `gen_exc.s` code checks for.

After the check for hardware interrupts, then the check for hardware and software exceptions begins. A check is made to see if this is a system call (`EXC_SYSCALL`). Then a check is made to see if this is TLB read miss (`EXC_MOD` or `EXC_RMISS`). TLB write misses are handled later in the logic (see the logic path on the diagram leading to `ecommon`, then `PDA`, then `VEC_tlbmiss`). If this is not a TLB read miss, then a check is made to see if we have a watch (`EXC_WATCH`) or breakpoint (`EXC_BREAK`) exception (these seem to be primarily pathways related primarily to debugging). Finally, all the remaining exceptions (software exceptions, all of which have a prefix of "SEX_C....", below) fall through the longway code, then to `ecommon`, then, to the appropriate handler.

```

+ Local register exception codes +
#define EXC_CODE(n)  (0 < n < 20)

+ Hardware exception codes +
#define EXC_INT      EXC_CODE(0)  * interrupt *
#define EXC_MOD     EXC_CODE(1)  * TLB mod *
#define EXC_RMISS   EXC_CODE(2)  * Read TLB Miss *
#define EXC_WMISS   EXC_CODE(3)  * Write TLB Miss *
#define EXC_RHDE    EXC_CODE(4)  * Read Address Error *
#define EXC_WHDE    EXC_CODE(5)  * Write Address Error *
#define EXC_ISE     EXC_CODE(6)  * Instruction Bus Error *
#define EXC_DBE     EXC_CODE(7)  * Data Bus Error *
#define EXC_SYSCALL EXC_CODE(8)  * SYSCALL *
#define EXC_BREAK   EXC_CODE(9)  * Breakpoint *
#define EXC_ILI     EXC_CODE(10) * Illegal Instruction *
#define EXC_CPD     EXC_CODE(11) * Coprocessor Unusable *
#define EXC_OVERFLOW EXC_CODE(12) * Overflow *
+ 0x4000 - 0x810000
#define EXC_TRAP    EXC_CODE(13) * Trap exception *
#define EXC_VCEI    EXC_CODE(14) * Viol. Cchexon on Inst. Fetch *
#define EXC_VCEI    EXC_CODE(15) * Viol. Cchexon on Inst. Fetch *

```


Module 7: Process Management Overview

Process Management Overview

This section provides an overview of IRIX processes and process management.

By the end of this section, the student will be able to:

- Describe the difference between a process and an executable file
- Use the `elfdump` tool to examine an executable file.
- Define "process" and describe a virtual process image.
- Describe a stack format, and the differences between a user stack and a kernel stack
- Use the `gmemusage` tool to display and examine system and process physical memory usage
- Describe some of the key structures in process control, and their functions
- Describe the flow of execution in a context switch
- Describe the flow of execution in a system call

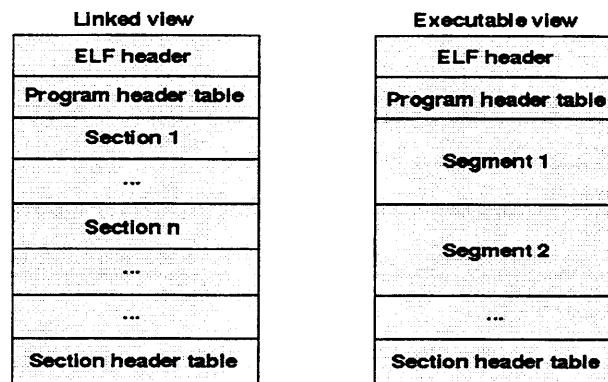
Process Management Overview

The process management facilities within IRIX are at the heart of the operating system. They are responsible for coordination of all the tasks invoked by users as well as system tasks. The process management subsystem's responsibility includes the following:

- User process life cycle
Creation, execution, interruption, and termination of user processes.
- CPU scheduling
All runnable processes are placed on run queues. Run queues must be constantly maintained with the highest priority processes scheduled to run next at any point in time.
- Context switching
Once a process gets connected to a CPU, process management must determine how long the process is allowed to use the CPU before the CPU switches to another process.
- Accounting
The kernel must keep track of how much execution time a process has consumed in user mode as well as kernel mode.
- Memory usage
The memory management subsystem must be consulted to allocate and deallocate main memory when a process is initiated, or when it expands or contracts in memory.
- Exception handling
When programs executing within processes cause exceptions, the operating system must notify the affected process via the signal mechanism.
- I/O processing

Processes need to associate themselves with files for purposes of reading and writing. The process management subsystem must coordinate with the file and I/O management subsystems.

Executable Files and Processes Diagram



Executable Files and Processes Diagram

A *process* is created from an executable file stored in the file system. Executable files generated by any compiling system are called *a.out* files, because the default name for the compiler and linkage editor output is *a.out*.

At the beginning of each *a.out* file is a header. The header contains the information about the format and structure of the program within the file. The header tells the system how to build a process in memory from the executable file stored on disk.

All *a.out* format files in IRIX use the format called the *Extensible Linking Format* (ELF). The diagram above shows an overview of the ELF format. The stored format of an *a.out* file is called the *linked view* (a program). As the file is loaded into memory to execute, the format of the *a.out* file within memory is called the *executable view* (a process).

ELF *a.out* files are constructed from various sections that are described within the header. The sections describe the organization of the parts of an executable file as stored on disk. Sections are used to hold parts of a program like instructions (code text), data, symbol tables, etc.

When an *a.out* file is loaded into memory for execution, three kinds of logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. A segment holds the parts of the program for the execution view and may contain one or more sections from the executable file.

A single-threaded program loaded into memory may have multiple text and data segments, but only one stack segment. The text segments are not writable by the program; if other processes are executing the same *a.out* file, the processes will share the same text segments.

7-4

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Executable Files and `elfdump(1)`

```
ELFDUMP                                     ELFDUMP

NAME
elfdump - dumps selected parts of a 32-bit or a 64-bit ELF object
file/archive and displays them in ELF style

SYNOPSIS
elfdump [ options ] file

DESCRIPTION
The elfdump command dumps selected parts of a given ELF object
file.

This command works for 32-bit or 64-bit ELF object files or ELF
archives only. It accepts these options and many others (see
man page):

...
-f Dumps the file (ELF) header.
-h Dumps all section headers in the file.
...
```

The `elfdump` command will display selected portions of an *a.out* executable file. The options to `elfdump(1)` control what portions are displayed. The examples on the following pages will only illustrate the use of the `-f` and `-h` options. For additional options, see the `elfdump(1)` man page.

7-5

22jul1998

TR-IKI rev 0.7b SGI Proprietary


```

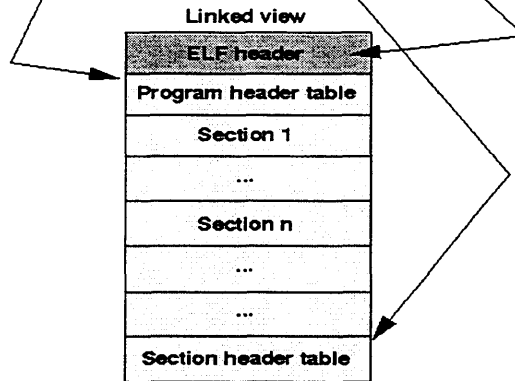
$ pwd
/sbin
$ ls -l cat
-rwxr-xr-x 1 root sys 70236 Jan 19 1997 cat
$ elfdump -f cat

```

```
cat:
```

**** ELF HEADER ****				
Class	Data	Type	Machine	Version
Entry	Phoff	Shoff	Emsize	Flags
Phentsize	Phnum	Shentsz	Strnum	Shstrndx
32-bit	2MESH	Exec	MIPS	CURRENT
0x10000130	0x34	0x1107c	0x34	NOREORDER MIPS2
0x20	4	0x28	12	11

```
$
```



```

$ elfdump -f cat
cat:

```

**** ELF HEADER ****				
Class	Data	Type	Machine	Version
Entry	Phoff	Shoff	Emsize	Flags
Phentsize	Phnum	Shentsz	Strnum	Shstrndx
32-bit	2MESH	Exec	MIPS	CURRENT
0x10000130	0x34	0x1107c	0x34	NOREORDER MIPS2
0x20	4	0x28	12	11

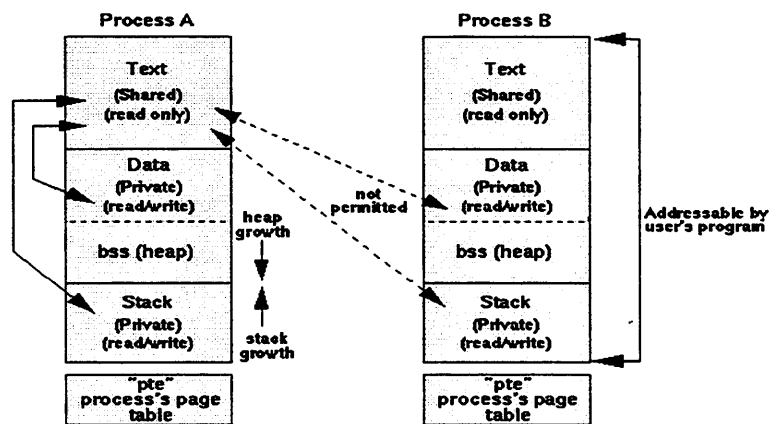
```
$ elfdump -h cat
```

```
cat:
```

**** SECTION HEADER TABLE ****							
[No]	Type	Link	Info	Addr	Offset	Size	Name
				Adralgn	Entsize	Flags	
[1]	SHT_MIPS_OPTIONS	0	0	0x10000b4	0xb4	0x60	.MIPS.options
				0x4	0	ALLOC NOSTRIP	
[2]	SHT_MIPS_REGINFO	0	0	0x10000114	0x114	0x18	.reginfo
				0x4	0x18	ALLOC	
[3]	SHT_PROGBITS	0	0	0x10000130	0x130	0xc60	.text
				0x10	0x1	ALLOC EXECINSTR	
[4]	SHT_PROGBITS	0	0	0x1000cd90	0xcd90	0x20	.init
				0x10	0x1	ALLOC EXECINSTR	
[5]	SHT_PROGBITS	0	0	0x1004d000	0xd000	0x21d0	.rodata
				0x10	0x1	WRITE ALLOC	
[6]	SHT_PROGBITS	0	0	0x1004f1d0	0xf1d0	0x1060	.data
				0x10	0x1	WRITE ALLOC	
[7]	SHT_PROGBITS	0	0	0x10050230	0x10230	0x38	.lit8
				0x10	0x8	WRITE ALLOC GPREL	
[8]	SHT_PROGBITS	0	0	0x10050270	0x10270	0x250	.sdata
				0x10	0x1	WRITE ALLOC GPREL	
[9]	SHT_NOBITS	0	0	0x100504c0	0x104c0	0x5c	.sbss
				0x10	0x1	WRITE ALLOC GPREL	
[10]	SHT_NOBITS	0	0	0x10050520	0x104c0	0x3624	.bss
				0x10	0x1	WRITE ALLOC	
[11]	SHT_STRTAB	0	0	0	0x11000	0x79	.shstrtab
				0	0		

```
$
```

Process Definition Diagram



Process Definition Diagram Explanation

A *process* is the execution of a program or executable file stored in the file system. An IRIX process is partitioned into several regions as shown above. All processes will have text, data, and stack regions but may contain others such as shared memory and memory mapped regions.

- Text

Contains a sequence of bytes that the CPU interprets as machine instructions. Has status "read only" and may be shared by multiple processes; that is, multiple processes may be executing concurrently all issuing instructions from the same shared text area. Because the text area can be shared, individual processes are not allowed to modify it.

- Data

A memory region private to the individual process and can be read or written by the process' instructions (text). Consists of two parts; an initialized area and uninitialized area usually referred to as the *bss* or *heap* area. Heap area can grow dynamically as the process needs more space. Heap growth is in the direction of the stack, ie, towards higher virtual addresses.

As shown above, a process cannot read or write to any other process' data or stack regions.

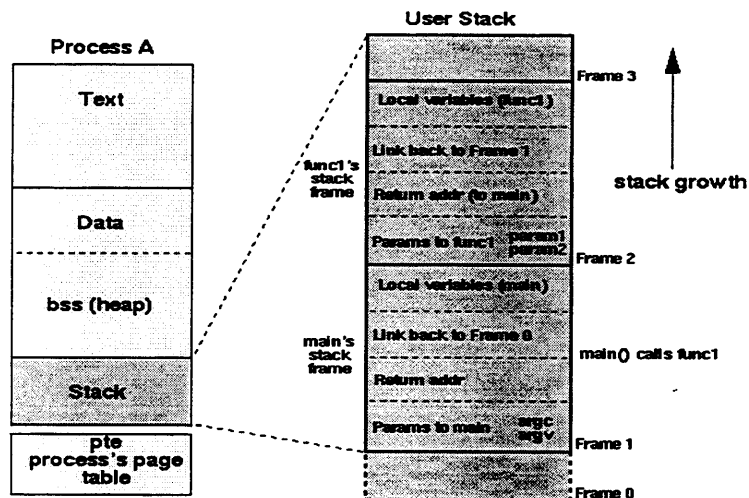
- Stack

Used to hold locally allocated variables and parameters passed to functions. Is automatically expanded as needed when process invokes functions or subroutines. Stack growth is in the direction of the heap, ie, towards lower virtual addresses. A process has two stacks. The *user stack* is used while executing instructions in user mode. The *kernel stack* is used for executing instructions in kernel mode or while the kernel is executing instructions "on behalf of" or "in the context of" the user process, such as when the kernel executes system call code which the user process requested.

As of IRIX 6.5, the user information traditionally stored in the "u area" on UNIX systems has been dispersed into other areas of the IRIX kernel. That information is now in various parts of the *uthread*, *kthread*, and *proc* structures.

The process's page table, the "pte", is used to load the TLB for the purpose of virtual to physical address translations.

User Stack Diagram



User Stack Diagram Explanation

Attributes of the *user stack* are:

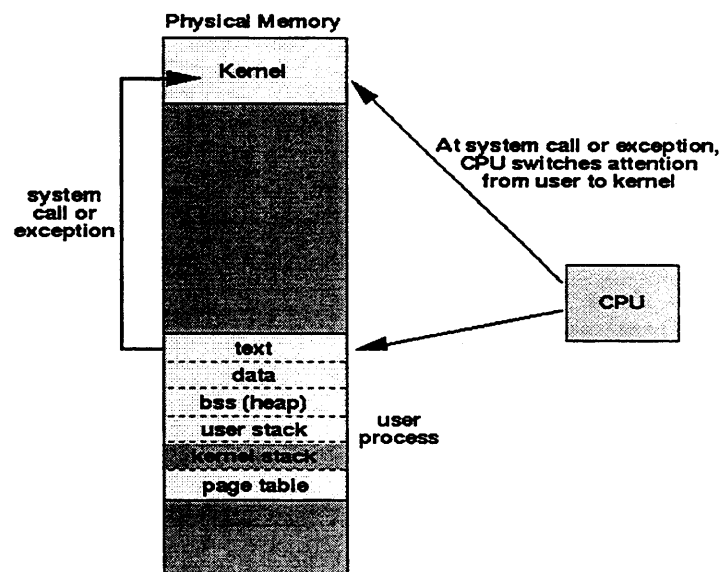
- Automatically created, and size dynamically adjusted, at run time.
- Logical stack frames that are pushed onto the stack when a function is called, and popped off the stack when returning.
- Stack pointer indicates current position in the stack.
- Stack frame contains parameters passed to the function, function's local variables, location of previous frame, return address to calling function,
- Kernel grows the stack as needed.

7-11

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Kernel Stack Diagram



7-12

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Kernel Stack Diagram Explanation

Because a process can execute in two modes, user or kernel, a separate stack is used for each mode. As stated on the previous page, the user stack contains the arguments and local variables for functions executing in user mode.

The *kernel stack* contains the stack frames for functions executing in the kernel in kernel mode. The function and data entries on the kernel stack refer to functions and data in the kernel, not the user program. The kernel stack's construction is the same as that of the user stack.

The "kernelstack" contains information about what the kernel is doing on behalf of a particular process. This information can help to determine the cause of a system panic or hang. The kernelstack virtual address is the same for all processes running on the system. For example, on an IP27 system, the address of the kernelstack is 0xffffffff800. The kernelstack address is platform specific and is determined when the kernel is built.

Information is contained in the proc struct about the mapping of the kernelstack address to a particular physical memory page.

*Kernel stacks are limited in size to 1 page
(or 2 pages for a 32-bit arch.)*

Processes and Kernel Threads

In IRIX revision 6.1 and earlier versions of IRIX, the process was the central mechanism for distributing processor resources over a collection of independent and cooperating tasks in the operating system. IRIX 6.2 introduced the migration toward the use of kernel threads (kthreads) as a central mechanism.

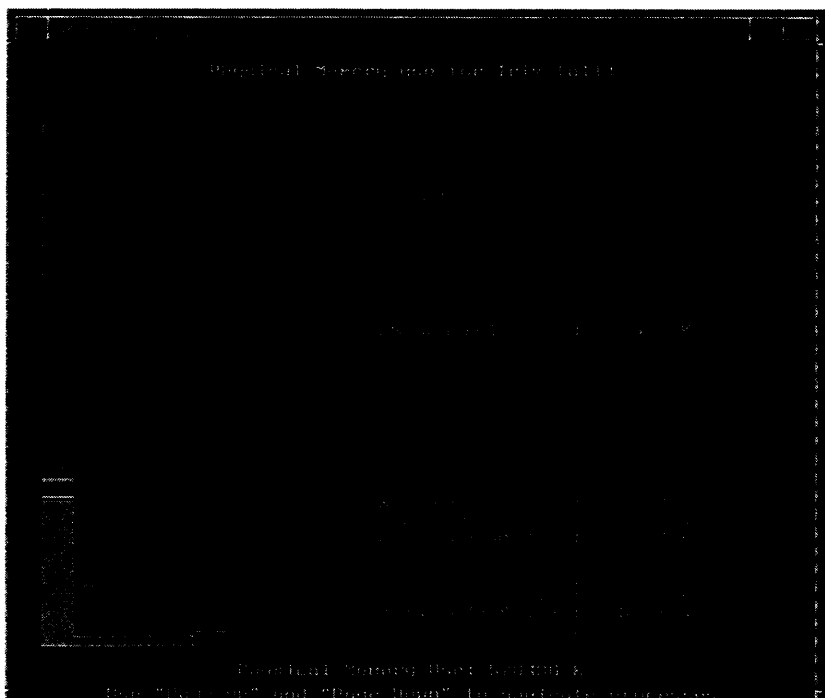
Kernel threads resemble the execution model of a UNIX process and consist of a code stream, private stack, and private register space. Unlike UNIX processes, kthreads are inexpensive to create and destroy, and can be quickly scheduled. A kthread has an associated user process context only if it is running on behalf of a system call or page fault, otherwise it has no logical connection to a user process.

With IRIX 6.2 and 6.3, only a partial conversion was made. The construct of a kthread was introduced, however it was still closely associated with entries in the proc table. In fact, a proc table entry was allocated for each active kthread in the system (even those that had no process context). Not until IRIX 6.4 was the conversion more or less complete (the evolution will continue with future revisions of IRIX). The kthread has now become the fundamental execution entity in the system. There are three types of kthreads:

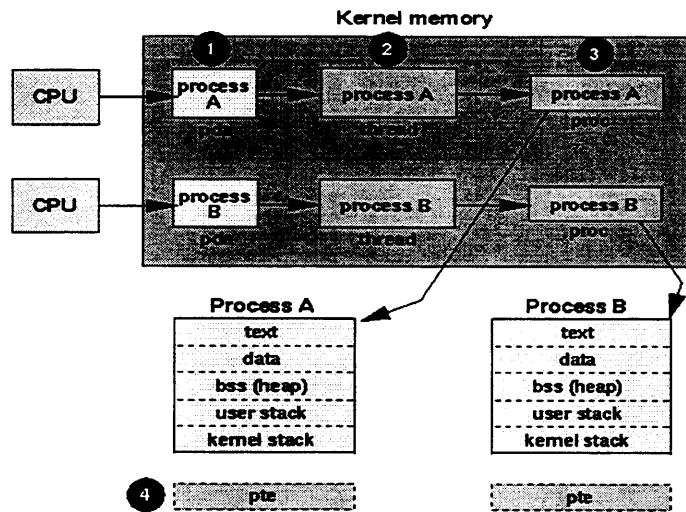
- User process
- Interrupt thread (ithread)
- Service thread (stthread)

about 500 g memusage

IRIX Physical Memory `gmemusage(1)` Display



Process Control Diagram



Process Control Diagram Explanation

This diagram provides a brief overview of how the kernel controls individual user processes.

1. *pda* structure

Each CPU has a *private data area (pda)* in main memory which points to the thread structure for the process currently connected to the CPU.

2. *thread* structure

When a process is created, a *thread* structure is dynamically allocated in the kernel which is used in controlling the process. The thread structure holds information such as status of signals, CPU scheduling information, and current system call and arguments passed.

3. *proc* structure

When a process is created, a *proc* structure is also dynamically allocated in the kernel which is used in controlling the process. The *proc* structure keeps track of who its parent, child, and sibling processes are as well as what process group it belongs to.

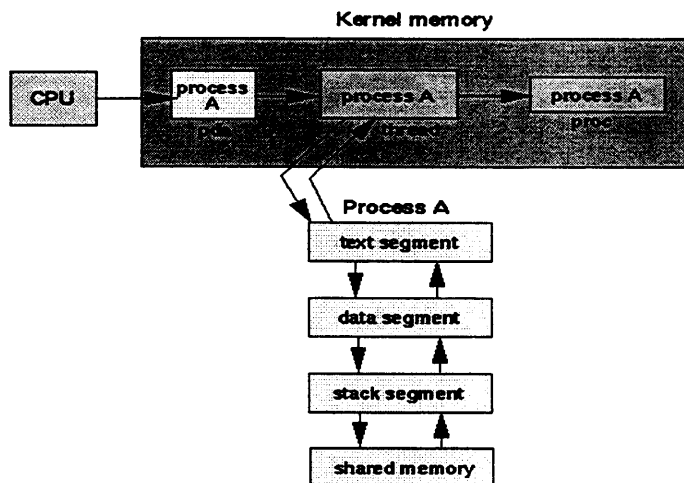
4. *pte* structure (page tables)

Helps map virtual process pages to physical memory pages.

Because the thread and *proc* structures are always resident in memory, the information maintained in these structures for a particular process is always available to the kernel, even if the process is paged out. Therefore, the thread and *proc* structures contain all of the data needed about a process even though the process might not be present in main memory.

Conversely, if a process is paged out by the kernel to gain space for a different process, the information in the user area is not available to the kernel until the process is paged back into memory. Therefore, the user area contains process control information that is not needed by the kernel when the process is inactive or paged out of main memory.

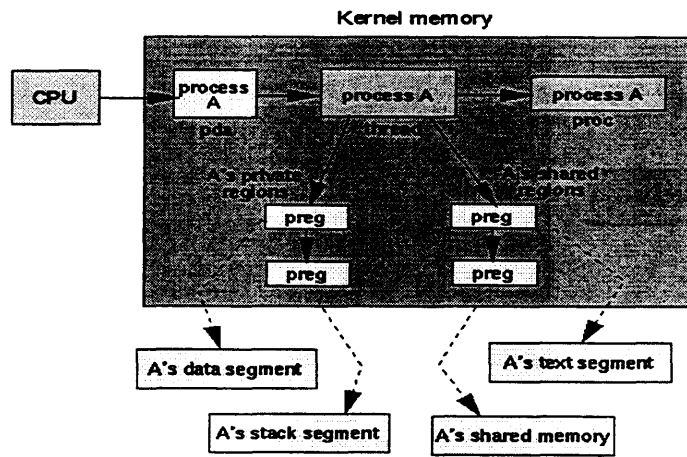
Process Segments or Regions



The IRIX kernel divides the virtual address space of a process into logical *segments* or *regions*. A segment or region is a contiguous area of virtual address space of a process which can be treated as a distinct object to be shared or protected. Several processes can share a segment or region.

For example, several processes may execute the same program, such as multiple users of the same shell program. Therefore, it makes sense for them to share the same copy of the text region. In a similar manner, several processes may cooperate by sharing a common shared-memory region. The process region mechanism also allows the kernel to protect regions of a process's address space so that the process itself cannot alter the region. This is done with the text region of a process.

Kernel's Region Tables Diagram



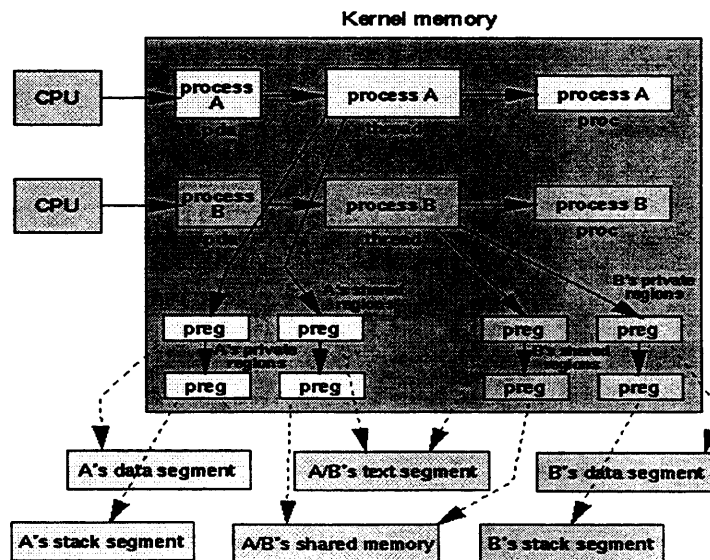
Kernel's Region Tables Diagram Explanation

The kernel maintains a *region table* (not shown above) and allocates an entry in the table for each active region on the system. Region table entries keep track of where each region resides in physical memory. Process-independent attributes are kept in the region table entries.

Each process also has a *per process region table* where each entry is usually referred to as a *pregion* (*preg* in the diagram). Each *preg* entry has a pointer to a region table entry which has pointers to where the region resides in physical memory (shown as a dashed arrow in diagram). The *preg* entry also contains a permission field that indicates the type of access allowed to the process: read-only, read-write, or read-execute. Process-specific attributes are kept in the region structure.

A process' *pregions* (*pregs*) are maintained in two separate lists. One list controls the regions which are considered private and the other controls those which are shared. The process' thread structure locates the private and shared *pregion* lists.

Region Sharing Diagram



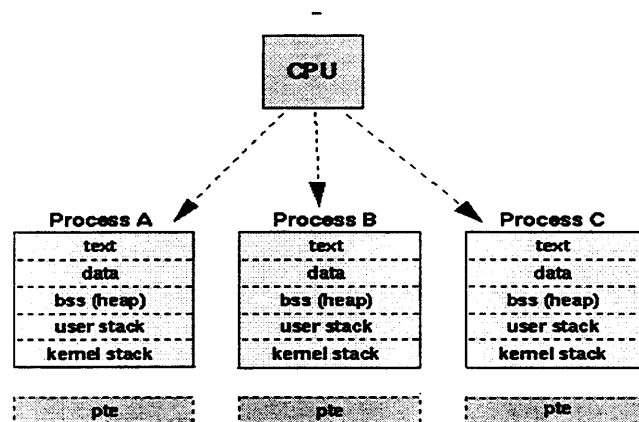
Region Sharing Diagram Explanation

Several processes can share parts of their address spaces via a common region. Each process sharing a region accesses the region via a private *pregion* (*preg*) entry.

The illustration above shows two processes (A and B) which are executing the same program and sharing a shared-memory region. The obvious advantages of region sharing are:

- Reduction in physical memory requirements when multiple processes are executing the same program. For example, there is significant reduction in physical memory requirements for a program like the shell program which has many concurrent users.
- Much less kernel paging is required with one copy of a program's text and multiple processes executing within the same text area.
- Process startup time is reduced if desired program text is already in memory.

Multiprocessing

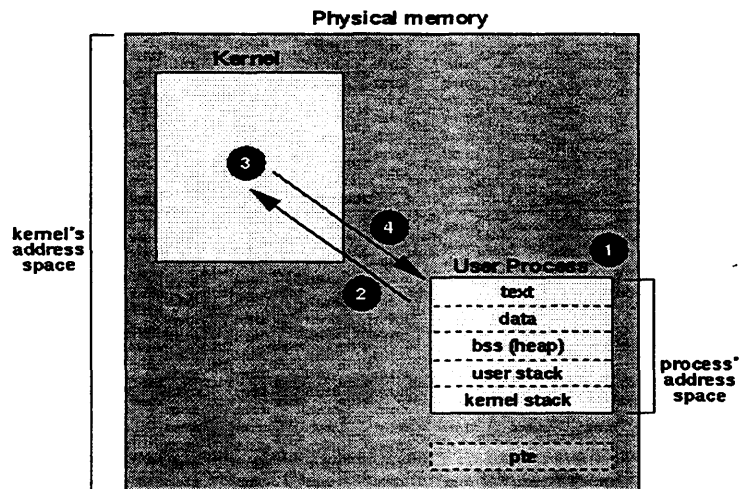


The IRIX system is a *multiprocessing* environment. Each CPU can execute in only one of two locations at any time: user process or operating system kernel. However, the operating system gives users the impression that a single CPU can give attention to multiple processes simultaneously, but only one user process can execute at a time per CPU when the CPU is not "in" the kernel.

The kernel provides this illusion by a mechanism called *time slicing*. Processes receive short bursts of CPU attention called time slices. In general, a single process will not receive 100% attention of a CPU. Therefore, to the user it looks like multiple processes are all executing simultaneously, but in reality it is just one process at a time executing in short bursts. The CPU(s) must switch attention to multiple processes (residing in priority order on the run queue) very rapidly to provide this illusion. Processes are switched in and out of a CPU typically every few milliseconds.

On a multi-CPU system, multiple processes execute simultaneously in the various CPUs.

Process Execution Flow Diagram



Process Execution Flow Diagram Explanation

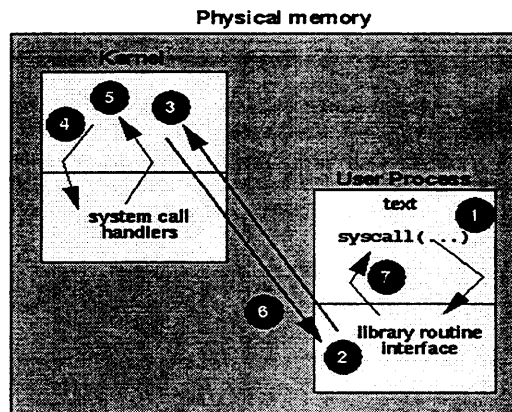
Each process runs in its own address space and behaves as if all of the machine resources are available for its exclusive use. The execution of multiple processes in parallel is achieved by switching different processes in and out of the CPU every few milliseconds.

The above diagram shows the typical flow of execution for a process.

1. When a CPU is executing in a user process, it is said to be operating in *user mode*. The process can only access memory that is within its address space.
2. When a user process makes a system call, generates an exception (fault), or when an interrupt occurs like a terminal interrupt (Ctrl-C) or a system clock interrupt, the user process's context is saved before the CPU executes kernel code.
3. When the CPU leaves the user process and enters the kernel, it is executing in the kernel on behalf of that user process and is said to be executing in *kernel mode*. In kernel mode, the CPU is authorized to execute privileged instructions and can access the code and data of any process. A user process cannot access kernel code, or the address space of any other process.
4. When the kernel has completed execution on behalf of the user process, it restores the context of the user process and returns CPU control to the process at the location where the user process was previously interrupted. Execution resumes in user mode.

From the viewpoint of a user program, a process' address space is a linear, flat, addressable area of memory starting at address zero and extending to a fixed address boundary set by both the hardware and operating system kernel. However, to the kernel, a process's address space is divided into discrete regions called text, data, heap (bss), and stack shown on previous pages.

System Call Interface Diagram



System Call Interface Diagram Explanation

A *system call* is the mechanism a user uses to invoke a function or perform a task in the kernel. There are hundreds of system calls a user can invoke; for example, `open(2)`, `read(2)`, `write(2)`, `close(2)`, `chmod(2)`, `chown(2)`, `kill(2)`, etc. The method used to invoke a system call and have control passed to the kernel for execution is illustrated above with explanation below:

1. The user's program issues a system call by specifying the name of the specific call with an attached list of arguments. For every possible system call supported in IRIX, there is a unique library routine which processes the system call request. Control is passed to this library routine.
2. From the viewpoint of the kernel, every system call is known by a unique integer which must be passed to the kernel for identification. The library routine invokes an instruction that changes the process' execution mode to kernel mode and causes control to be passed to the kernel's system call handling code passing along the integer identifier for the desired system call.
3. The kernel's system call handling code receives the integer to identify the system call the user is invoking, and looks up the system call number in a table (`sysent`) to find the address of the appropriate system call handler.
4. Control passes to the identified system call handler and the system call executes.
5. When finished with the user's request (for example, a file read request), the system call handler returns to the kernel code from which it was called.
6. The kernel returns to the user process' library routine which originally passed control to the kernel switching back to user mode.
7. The library routine returns to the user's program where the system call was made with a return value indicating success or failure.

Module 8: IRIX System Calls

IRIX System Call Processing

Unit covers:

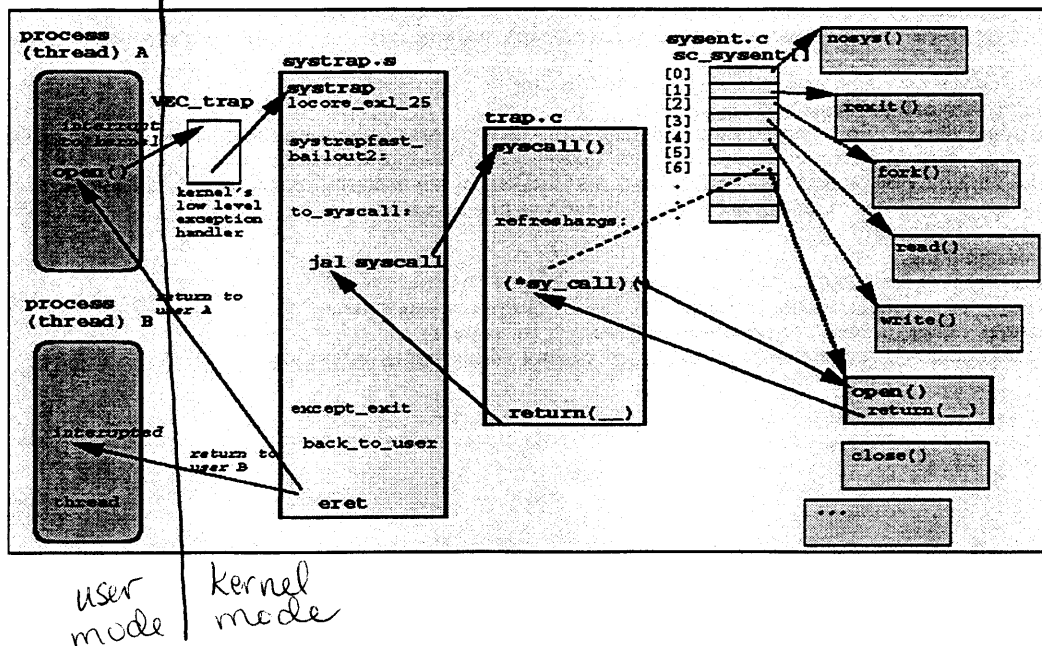
- Overview of IRIX kernel system call processing
- List and role of key components in system call processing
- System call walk-through
- System call argument processing including return value conventions
- System call *icrash*(1m) examples

System Call Review

- User processes access Operating System Services via a mechanism called **System Calls**.
- System calls are performed with these general steps:
 1. User program prepares **calling argument** values according to each one's prototype as described in the calls `man(2)` page.
 2. Program performs a hardware instruction that causes in interrupt. The mnemonic for this in IRIX is **syscall**.
 3. The interrupt is "trapped" by the OS; the OS performs the operation (if legal) and returns one or more **return values** as described in the call's `man(2)` page. See `man` page for `intro(2)` for list of standard error return values.
 4. OS returns control to the user process, passing any **return values** to the user program.
 5. The user program checks **return values** for errors and handles error or continues processing.

- Programs have access to System Calls with several methods:
 1. User codes the above "calling sequence" in program using direct assembler code.
 2. User uses standard UNIX system call library routines (a.k.a. `open(2)`, `read(2)`, `close(2)`) to directly access the system call.
 3. User uses standard "higher level" library functions to access system services. These library routines take a lot of the clerical work out of accessing system services or provide services in themselves. For instance:
 - `fopen(3)`, `fclose(3)`, `fread(3)`, and `fwrite(3)` provide file I/O with user library level (double) buffering and other services.
 - `psignal(3)` provides general access to kernel signal processing services.
 - `malloc(3)` allocates and manages user (heap / BSS) memory, making `break(2)` system calls to request that the kernel expand user memory.
 4. Compilers generate System Call sequences as part of their command support.
 5. Compiler commands such as FORTRAN's `OPEN` and `READ` build upon library `man(2)` system call routines.
 6. Most compilers provide direct access to both `man(2)` and `man(3)` library routines.
 7. Many compilers allow imbedding of assembler commands within their "normal" code. These assembler routines may make system calls as described above.
- In all cases, the same low level kernel system call is invoked and processed by the OS. This unit describes kernel system call processing at this level.

System Call Component Diagram



System Call Overview

The diagram shows the general flow of control for IRIX system call processing. The `open(2)` system call is used as a typical example.

1. User process "A" executes `open(2)` system call.
The system call library code invokes a `syscall` interrupt.
Control switches to the kernel.
2. Low level kernel exception handler (trap) routines decode the exception type and dispatch control to assembler routine `systrap` for system call exceptions.
3. Routine `systrap` check for usage errors (e.g. interrupt must be from user, not kernel) and save user's CPU register context.
Systrap fabricates a kernel stack and calls C function `syscall()`.
4. Function `syscall()` accesses user's calling arguments.
 - A check is made to make sure that the system call number fits within the scope of the `sysent []` table.
 - The system call number argument is used to load the corresponding interrupt function address from the kernel `sysent []` table.
 - The `sysent []` table argument limit is used to check the caller's number of arguments.
 - If errors, `syscall()` returns to user with a standard error code (`errno.h`).
 - If no error, `syscall()` calls kernel function corresponding the system call number.

5. Processing continues at the kernel system call function.
 - During call processing, error conditions may cause the function to return to `syscall()` without completing the required work. `syscall()` returns the error (`syserr.h`) code to the user process.
 - The thread of logic started in the system call may block (sleep), waiting for some condition (resource) in the kernel.
This is called a "context switch" and is covered in another unit.
 - Eventually the system call processing completes.
 - Functions pass one or two "return values" to the calling user process indicating the success or failure of the call. The values are documented as the call's RETURN VALUES in the `man(2)` pages.
 - Additional data may be passed between the user and the kernel via user calling argument addresses; for example the path address in an `open(2)` or buffer address in a `read()`.
6. Returning to `syscall()` the kernel:
 - Does final error checking.
 - Restores the calling process's (A) CPU register context -OR-
 - Calls `soft_trap()` to schedule another process (B), restoring it's context instead.
 - Process A resumes either immediatly or when it is resumed by `swtch()`.
 - Process A should check it's return value(s) before proceeding with other program logic.

System Call Walk Through

User Makes System Call

Calling arguments are loaded into CPU registers. By IRIX convention, these are the "a" registers starting with reg a0. The man page for `open(2)` shows C SYNOPSIS:

```
int open (const char *path, int oflag, ... /* mode_t mode */)

```

For this C `open` command : `int fd = open("/tmp/opensam",O_CREAT,0700);`

- The address of the path string literal `"/tmp/opensam"` is loaded into register `a0`.
- The open flag `O_CREAT` (value `0x100`) is loaded into register `a1`.
#define O_CREAT 0x100 /* open with file create (uses third open arg) */
- The mode value `0700` is loaded into register `a2`.
- Other operands may be used as defined in the `open(2)` man page.
- The program calls the open system call library function `open()` in `open.s` which loads register `v0` with the system call number.

The number for each call is defined in file `sys.s`. This number represents a position (index) into a system call entry points table called `sysent[]`. For `open(2)`, this is `1005` (the base of the table is `1000`).

- The open library code executes the `syscall` machine instruction causing an interrupt into the kernel.
- Register content summary:
 - `a0-a2` contain the calling arguments.
 - `v0` contains the system call number.

Sample assembler code for open(2)

```
9: main() { from sample program opn.c
10:
[ 10] 0x10000b38: 8f 84 80 1c      lw      a0,-32740(gp)  local memory pages
[ 10] 0x10000b3c: 24 84 10 00      addiu   a0,a0,4096    path string literal
[ 10] 0x10000b40: 24 05 01 00      li      a1,256        O_CREAT=0x100
[ 10] 0x10000b44: 24 06 01 c0      li      a2,448        mode=0700
[ 10] 0x10000b48: 8f 99 80 30      lw      t9,-32720(gp) &open()
[ 10] 0x10000b4c: 03 20 f8 09      jalr   ra,t9         call open -> _open64

_open64: from libc.so.1
[ 18] 0xfa3e4a0: 24 02 03 ed      li      v0,1005      index into sysent table
[ 18] 0xfa3e4a4: 00 00 00 0c      syscall
[ 18] 0xfa3e4a8: 14 e0 00 03      bne    a3,zero,0xfac5078 -> _cerror (indirectly)
[ 18] 0xfa3e4ac: 00 00 00 00      nop
[ 19] 0xfa3e4b0: 03 e0 00 08      jr     ra            -> back to main()

_cerror: from cerror.s
[ 30] 0xfa384f8: 3c 0e 00 12      lui    t2,0x12
[ 30] 0xfa384fc: 65 ce 6f 58      daddiu t2,t2,28504
[ 30] 0xfa38500: 01 d9 70 2d      daddiu t2,t2,t9
[ 31] 0xfa38504: 8d c1 93 f8      lw     at,-27656(t2)  at=&errno (global error)
[ 31] 0xfa38508: ac 22 00 00      sw     v0,0(at)      save v0 in global errno
[ 34] 0xfa3850c: 8d cc 93 a8      lw     t0,-27736(t2)
[ 35] 0xfa38510: 8d cd 93 f8      lw     t1,-27656(t2)
[ 34] 0xfa38514: 8d 8c 00 00      lw     t0,0(t0)
[ 36] 0xfa38518: 11 8d 00 02      beq   t0,t1,0xfaf484c
[ 36] 0xfa3851c: 00 00 00 00      nop
[ 37] 0xfa38520: ad 82 00 00      sw     v0,0(t0)     save v0 in per/ thread errno
[ 41] 0xfa38524: 03 e0 00 08      jr     ra            -> back to main()

main() continued
[ 10] 0x10000b50: 00 00 00 00      nop
[ 10] 0x10000b54: 8f 85 80 40      lw     a1,-32704(gp)  &fd
[ 10] 0x10000b58: ac a2 00 00      sw     v0,0(a1)      fd=v0
11: fd = open("/tmp/opensam",O_CREAT,0700);

```

Kernel Traps the Interrupt

- Low level interrupt handler routines test interrupt (exception) code, calling **systrap** (`systrap.s`) for the **syscall** hardware exception.
- Assembler code in **systrap**
 - Saves CPU registers in user process's `uthread` exception frame area.

Calling argument values from the `a` registers are now in the process thread area.

- Switches system times (accounting) over to OS.
- Use system call number (`sysnumber`) in `v0` as index into the **sysent**[] table to access kernel function address, call argument count, and call flags.
- C function **syscall()** is called to dispatch the system call.

Syscall() Dispatches the Call

- System call counts are incremented; by call number and total.
- The system call number and argument values are checked for sanity (Specific checking of argument values is done by each system call function.) Errors are returned as described below.
- User system call arguments are copied (up to 8 of them) to the `uthread ut_scallargs`[] array.
- The specific system call function is called as defined in the kernel **sysent**[] table.

Kernel Performs Specific System Call

- Kernel executes functions as initiated by the "top level" system call function
For example: `open()` calls `copen()` which may call `xfsoopen()`, and so on.
- Very often the kernel performing a system call must wait for some resource such as an I/O operation. In this case:
 - The function calls `sleep()` (or a variant of `sleep()`) to give the CPU to another process (thread).
 - The CPU selects another process thread and resumes processing that one.
 - Eventually the event the process was waiting for (e.g. I/O) completes. The interrupt handler for that event awakens this sleeping process with a `wakeup()` call.
 - The kernel will eventually select this process to resume where it left off.
 - A system call may result in many sleep-wakeup situations before it completes.
- If successful, "top level" functions return specific RETURN values as defined in the system call `man(2)` page.
- Kernel functions check for errors, returning specific error codes as described below.
- In any case, control returns to the "top level" system call function, which returns to `syscall()`.

syscall() Resumes Processing

- If system call return an error the error:
 - The error is posted to the user (set in `errno`).
 - The process may be sent a signal .
- If no error:
 - Set user return values `rv1` and `rv2` into registers `v0` and `v1` (see argument processing below).
- Clear flags indicating that the system call is finished.
- Return to `sysstrap()`.

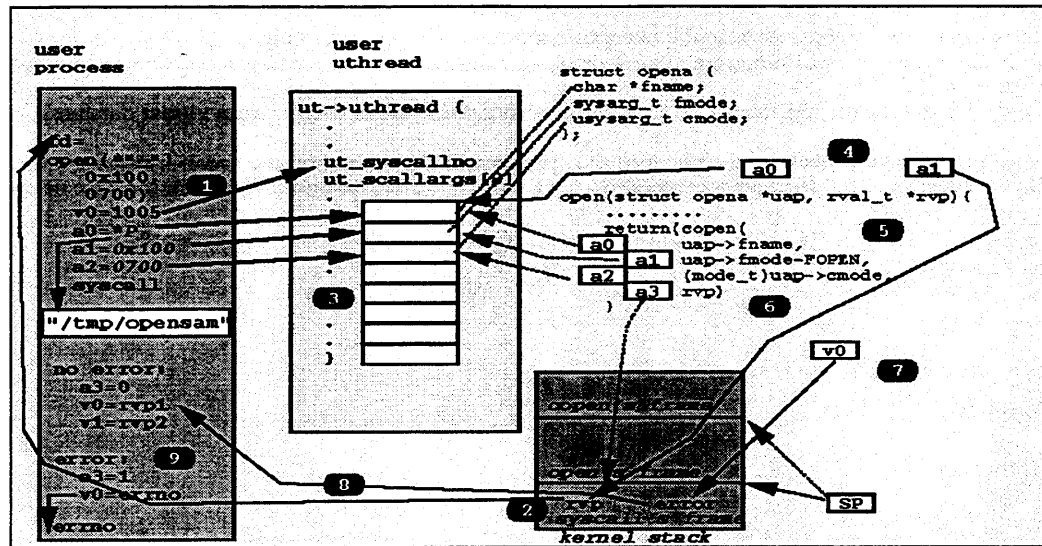
Systrap() Resumes Processing

- Kernel system call functions may set a **resched** flag in the kernel when they perform some operation that changes the potential scheduling of processes (thread) in the system. Examples:
 - Fork creates a new process which may need to run.
 - Exit destroys a process leaving the CPU free to run another.
 - A signal is sent to a process awakening it from sleep.
 - I/O completes awakening a process.
- When the **resched** flag is set, function **qswtch()** is called to check a process run queue. The most worthy process is selected to resume. (See process scheduling for more detail).
- The kernel system call timer is stopped and the user timer is resumed.
- The first user stack **TLB** is loaded.
- The user's CPU context (register values) are restored.
- **ERET** machine instruction is executed. Control resumes at the address in **EPC**, which was the user **PC** at the time of the syscall exception interrupt.

User Resumes Processing

- The library routine returns control to the calling user function.
- The user SHOULD check the return value (**v0**) and **errno** for error before continuing.
- In the open sample used in this walk-through, the integer **fd** receives the return value. If there was no error, **fd** will be an index to the user's open file descriptor in their open file table, the product of the **open(2)** system call operation.

System Call Argument Processing



System Call Argument Processing

Typical system call argument processing is shown using the `open(2)` system call as an example.

1. Open library code loads the calling argument values into registers `a0`, `a1`, and `a2`. The library code calls the kernel with the `syscall` command. On entry to the kernel, all user registers are save in the `uthread` exception frame. `Systrap()` creates a stack frame for the call to `syscall()` with room for `syscall`'s local variables such as `error` and `rvp`.
2. The system call number is placed in the `ut_syscallno` field and the user arguments are copied into the `ut_scallargs[8]` array, both in the process's `uthread` area.
3. Before `syscall()` calls the specific system call function, the kernel sets `a0` to point to the system call arguments, now in the `uthread` area. Register `a1` is set to point to the return value pointer `rvp` (in the stack). Register `a3` contains the system call number (not always used).
4. The called function used `a0` to locate the calling arguments. C code "maps" these arguments to their function use as seen by the `opena` structure in the `open(2)` system call.
5. Kernel function `open()` calls `copen()` passing the caller's argument (values and pointers) to it in the `a` registers as shown.
6. The system call function proceeds, performing each one's service until it finishes or returns with error.

7. At some time before returning, the system call functions store results in `rv1` and possibly `rv2`. These values are also in registers `v0` and `v1`. By convention, the system call functions return 0 (zero) if no error or non-zero if error to `syscall()`.
8. `syscall()` places the `error`, `v0`, and `v1` values in the exception frame `a3`, `v0`, and `v1` fields. `Systrap()` reloads the CPU registers (including the return vaalues) from the exception area just before returning to the interrupted user process.
9. The system call library routine checks register `a3` for non-zero, storing the value in user process global data area `errno`. Register `v0` is delivered to the user process as the result of the system call. Depending on the specific system call, the user can test this for success or failure, and access `errno` for a more specific reason for failure in the case of an error.

icrash(1M) Samples

Process uthread Display

```
>> proc -f a8000002014fcc00
      PROC ST  PID  PPID  PGID  UID          WCHAN NAME
-----
a8000002014fcc00  1    27    1    27    0 a8000002006b19d8 xlv_plexd

SELECTED FIELDS FROM THE KTHREAD STRUCT AT 0xa800000201508c00:
K_FLAGS(0x240020)=KT_SLEEP|KT_HOLD|KT_WSV
K_W2CHAN=0x0,K_STACK=0xfffffffff8000, K_STACKSIZE=16384
K_PRTN=1, K_PRI=-3, K_BASEPRI=-3, K_SQSELF=0, K_ONRQ=-1
K_SONPROC=-1, K_BINDING=-1, K_MUSTRUN=-1
K_LASTRUN=4, K_CPUSET=1, K_EFRAME=0x0, K_LINK=0x0
K_INHERIT=0x0, K_INDIRECTWAIT=0x0
K_RFLINK=0xa800000201508c00, K_RBLINK=0xa800000201508c00
K_FLINK=0xa800000201508c00, K_BLINK=0xa800000201508c00

SELECTED FIELDS FROM THE PROC STRUCT:
P_CHILDPIDS=0x0, P_SLINK=0x0, P_SHADDR=0x0

OPEN FILES FOR PROC 0xa8000002014fcc00:
      FD          FILE          RCNT          DATA          BH          FLAGS
-----
0  a8000002006a0240  3  a8000000009e5500  a8000002006a8418  3
1  a8000002006a0240  3  a8000000009e5500  a8000002006a8418  3
2  a8000002006a0240  3  a8000000009e5500  a8000002006a8418  3
=====
1 active processes found
```

uthread Detail

```
>> px *(uthread_t *)0xa800000201508c00
struct uthread_s {
  ut_kthread = kthread_t {
    k_regs = {
      [0] 0xa800000201508c00
      [1] 0x80
      [2] 0xa800000201508c84
      [3] 0x80
      [4] 0x0
      [5] 0x0
      [6] 0x100197d8
      [7] 0x10019798
      [8] 0xffffffffffffb940
      [9] 0x0
      [10] 0xc00000000200490
      [11] 0x68a1
      [12] 0x0
    }
    k_id = 0x10000005a
  }
  (edited)
  ut_syscallno = 0x5                                adjusted down from 1005, indexes to open()
  ut_scallargs = {
    user address of path
    open flags (is zero this case)
    open mode (is zero this case)
    [0] 0x10019138
    [1] 0x0
    [2] 0x0
    [3] 0x0
    [4] 0x0
    [5] 0x200e6c
    [6] 0x0
    [7] 0x200e70
  }
  (edited)
  ut_rsa_runable = 0x0
  ut_rsa_npgs = 0x0
  ut_rsa_locore = 0x0
  ut_rsa_pad = ""
}
```

Trace of open(2) System Call

```
>> trace a8000002014fcc00
=====
STACK TRACE FOR UTHREAD 0xa800000201508c00 (xlv_plexd, PID=27):
1 swtch[./os/swtch.c: 1086, 0xc00000000200490]
2 thread_block[./os/ksync/mutex.c: 159, 0xc00000000017b2f4]
3 sv_queue[./os/ksync/mutex.c: 1394, 0xc00000000017c998]
4 sv_timedwait_sig[./os/ksync/mutex.c: 1968, 0xc00000000017d6a0]
5 sv_wait_sig[./os/ksync/mutex.c: 1252, 0xc00000000017c650]
6 fifo_open[./fs/fifofs/fifovops.c: 118, 0xc000000000355ed4]
7 vn_open[./os/vnode.c: 1841, 0xc0000000001b8d50]
8 copen[./os/vncalls.c: 211, 0xc0000000001cb728]
9 open[./os/vncalls.c: 145, 0xc0000000001cb5bc]
10 syscall[./os/trap.c: 2737, 0xc00000000018cba4]
11 systrap[./ml/LOCORE/systrap.s: 314, 0xc00000000037c48]
r0/zero:0000000000000000 r1/at:fffffffffffffc0 r2/v0:00000000000003ed
r3/v1:0000000000000000 r4/a0:0000000010019138 r5/a1:0000000000000000
r6/a2:0000000000000000 r7/a3:0000000000000000 r8/a4:0000000000000000
r9/a5:0000000000200e6c r10/a6:0000000000000000 r11/a7:0000000000200e70
r12/t0:000000001001b010 r13/t1:000000001001b008 r14/t2:0000000000000008
r15/t3:000000003fff0000 r16/s0:0000000010019138 r17/s1:00000000100197b0
r18/s2:0000000000000000 r19/s3:0000000000000000 r20/s4:0000000000000000
r21/s5:000000000fb4f950 r22/s6:00000000100197d8 r23/s7:0000000010019798
r24/t8:0000000000000003 r25/t9:0000000010004494 r26/k0:0000000000000000
r27/k1:ffffffff8400ffb3 r28/gp:000000000fb58134 r29/sp:000000007fff29a0
r30/s8:0000000000000000 r31/ra:000000000fb08744 EPC:000000000fb01474
CAUSE=8, SR=ffffffff8400ffb3, BADVADDR=10004494
=====
```

not saved for
syscall exception
(unless DEBUG kernel)

only meaningful if
CAUSE of exception is TLBMISS

Trace Detail (partial)

```
>> trace -f a8000002014fcc00
=====
STACK TRACE FOR UTHREAD 0xa800000201508c00 (xlv_plexd, PID=27):
(edited)
8 copen[./os/vncalls.c: 211, 0xc000000001cb728]
  RA=0xc000000001cb5c4, SP=0xffffffffffffbe30, FRAME SIZE=128
  ffffffffbe30: a8000002006a0270 0000000301419c18
  ffffffffbe40: a800000201508c00 a800000201508c84
  ffffffffbe50: a800000201508c00 000000010019138
  ffffffffbe60: 0000000000000000 0000000000000000
  ffffffffbe70: 0000000000000001 ffffffffbed8
  ffffffffbe80: c000000001cb5c4 c00000001418d30
  ffffffffbe90: 000000000020000 000000000020000
  ffffffffbea0: a800000201508ea8 c0000000018cbac
9 open[./os/vncalls.c: 145, 0xc000000001cb5bc]
  RA=0xc0000000018cbac, SP=0xffffffffffffbeb0, FRAME SIZE=32
  ffffffffbeb0: c0000000018cbac 0000000000000000
  ffffffffbec0: c0000000018d374 a800000201508c00
```

8-21

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
10 syscall[./os/trap.c: 2737, 0xc0000000018cba4]
  RA=0xc00000000037c50, SP=0xffffffffffffbed0, FRAME SIZE=160
  ffffffffbed0: 000000007fff2fc0 0000000000000000
  ffffffffbee0: 0000000000000000 a800000201509368
  ffffffffbef0: 0000000000000000 0000000000000000
  ffffffffbf00: 0000000000000004 0000000000000005
  ffffffffbf10: 0000000000000000 a800000201509380
  ffffffffbf20: ffffffff8400ffb3 000000000000ffa0
  ffffffffbf30: 0000000000000000 c00000000037c50
  ffffffffbf40: 0000000100197b0 000000010019798
  ffffffffbf50: 00000000fb4f950 0000000000000000
  ffffffffbf60: 0000000000000b4 c0000000001a09c
11 systrap[./ml/LOCORE/systrap.s: 314, 0xc00000000037c48]
  r0/zero:0000000000000000 r1/at:ffffffffffffc0 r2/v0:00000000000003ed
  r3/v1:0000000000000000 r4/a0:000000010019138 r5/a1:0000000000000000
  r6/a2:0000000000000000 r7/a3:0000000000000000 r8/a4:0000000000000000
  r9/a5:000000000200e6c r10/a6:0000000000000000 r11/a7:000000000200e70
  r12/t0:00000001001b010 r13/t1:00000001001b008 r14/t2:0000000000000008
  r15/t3:000000003fff0000 r16/s0:000000010019138 r17/s1:0000000100197b0
  r18/s2:0000000000000000 r19/s3:0000000000000000 r20/s4:0000000000000000
  r21/s5:00000000fb4f950 r22/s6:0000000100197d8 r23/s7:000000010019798
  r24/t8:0000000000000003 r25/t9:000000010004494 r26/k0:0000000000000000
  r27/k1:ffff8400ffb3 r28/gp:00000000fb58134 r29/sp:00000007fff29a0
  r30/s8:0000000000000000 r31/ra:00000000fb08744 EPC:00000000fb01474
  CAUSE=8, SR=ffff8400ffb3, BADVADDR=10004494
=====
```

8-22

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Frame For open()

```
9 open[..os/vncalls.c: 145, 0xc000000001cb5bc]
RA=0xc0000000018cbac, SP=0xffffffffffffbe0, FRAME SIZE=32
ffffffffffffbe0: c0000000018cbac 0000000000000000 0(sp)=ra
ffffffffffffbec0: c00000000018d374 a8000000201508c00

>> findsym c0000000018cbac
=====
0xc0000000018cbac --> syscall + 0x3ac
=====
1 symbol found
```

Disassembly Code For open()

```
>> dis open 30
=====
[open:145, 0xc000000001cb590]      ori    a4,zero,0xffff
[open:145, 0xc000000001cb594]      lw     a2,20(a0)  a0=uap  a0+20 is mode
[open:144, 0xc000000001cb598]      daddiu sp,sp,-32
[open:145, 0xc000000001cb59c]      dsll  a4,a4,16
[open:144, 0xc000000001cb5a0]      move  a3,a1
[open:145, 0xc000000001cb5a4]      ld    a1,8(a0)  a0=uap  a0+8 is flags
[open:145, 0xc000000001cb5a8]      ori  a4,a4,0xfffe
[open:145, 0xc000000001cb5ac]      sd   ra,0(sp)
[open:145, 0xc000000001cb5b0]      nor  a4,a4,zero
[open:145, 0xc000000001cb5b4]      ld   a0,0(a0)  a0=uap  a0+0 is *fname
[open:145, 0xc000000001cb5b8]      daddu a1,a1,a4
[open:145, 0xc000000001cb5bc]      jal  0xc000000001cb600 (copen)
[open:145, 0xc000000001cb5c0]      sll  a1,a1,0
[open:145, 0xc000000001cb5c4]      ld   ra,0(sp)
[open:145, 0xc000000001cb5c8]      jr   ra
(edit)
=====
```

Frame For fopen()

```
8 fopen[../os/vncalls.c: 211, 0xc000000001cb728]
RA=0xc000000001cb5c4, SP=0xffffffffffffbe30, FRAME SIZE=128
ffffffffffffbe30: a8000002006a0270 0000000301419c18 (0/8)
ffffffffffffbe40: a800000201508c00 a800000201508c84 (16/24)
ffffffffffffbe50: a800000201508c00 0000000010019138 (32/40) (*fname)
ffffffffffffbe60: 0000000000000000 0000000000000000 (48/56) (cnode/)
ffffffffffffbe70: 0000000000000001 ffffffffbed8 (64/72) (fmode/*rvp)
ffffffffffffbe80: c000000001cb5c4 c00000001418d30
ffffffffffffbe90: 000000000020000 000000000020000
ffffffffffffbea0: a800000201508ea8 c0000000018cbac
```

Disassembly Code For kernel fopen()

```
>> dis fopen 20
-----
[copen:168, 0xc000000001cb600] daddiu sp,sp,-128
[copen:168, 0xc000000001cb604] sd a0,40(sp) a0=*fname
[copen:168, 0xc000000001cb608] sd a2,48(sp) a2=cmode
[copen:168, 0xc000000001cb60c] sd a3,72(sp) a3=*rvp
[copen:174, 0xc000000001cb610] move v0,a1
[copen:174, 0xc000000001cb614] sd a1,64(sp) a1=fmode
[copen:175, 0xc000000001cb618] andi at,a1,0x3
[copen:181, 0xc000000001cb61c] li v0,22
[copen:181, 0xc000000001cb620] sd s0,32(sp)
[copen:181, 0xc000000001cb624] sd zero,56(sp)
[copen:175, 0xc000000001cb628] beq at,zero,0xc000000001cb758
[copen:168, 0xc000000001cb62c] move s0,a1
[copen:186, 0xc000000001cb630] daddiu a2,sp,8
[copen:186, 0xc000000001cb634] daddiu a1,sp,0
[copen:181, 0xc000000001cb638] li a4,132
[copen:181, 0xc000000001cb63c] andi a3,s0,0x84
[copen:181, 0xc000000001cb640] xor a3,a3,a4
[copen:184, 0xc000000001cb644] li a4,-5
[copen:186, 0xc000000001cb648] lui a0,0x1
[copen:184, 0xc000000001cb64c] and a4,s0,a4
-----
```

Register Aliases

\$pc - current user pc
 \$sp - current value of stack pointer
 \$rn - register n
 \$fn - single precision floating point register
 \$dn - double precision floating point register
 \$mmhi - most significant multiply/divide result register
 \$mmlo - least significant multiply/divide result register
 \$fcsr - floating point control and status register
 \$feir - floating point exception instruction register
 \$cause - exception cause register

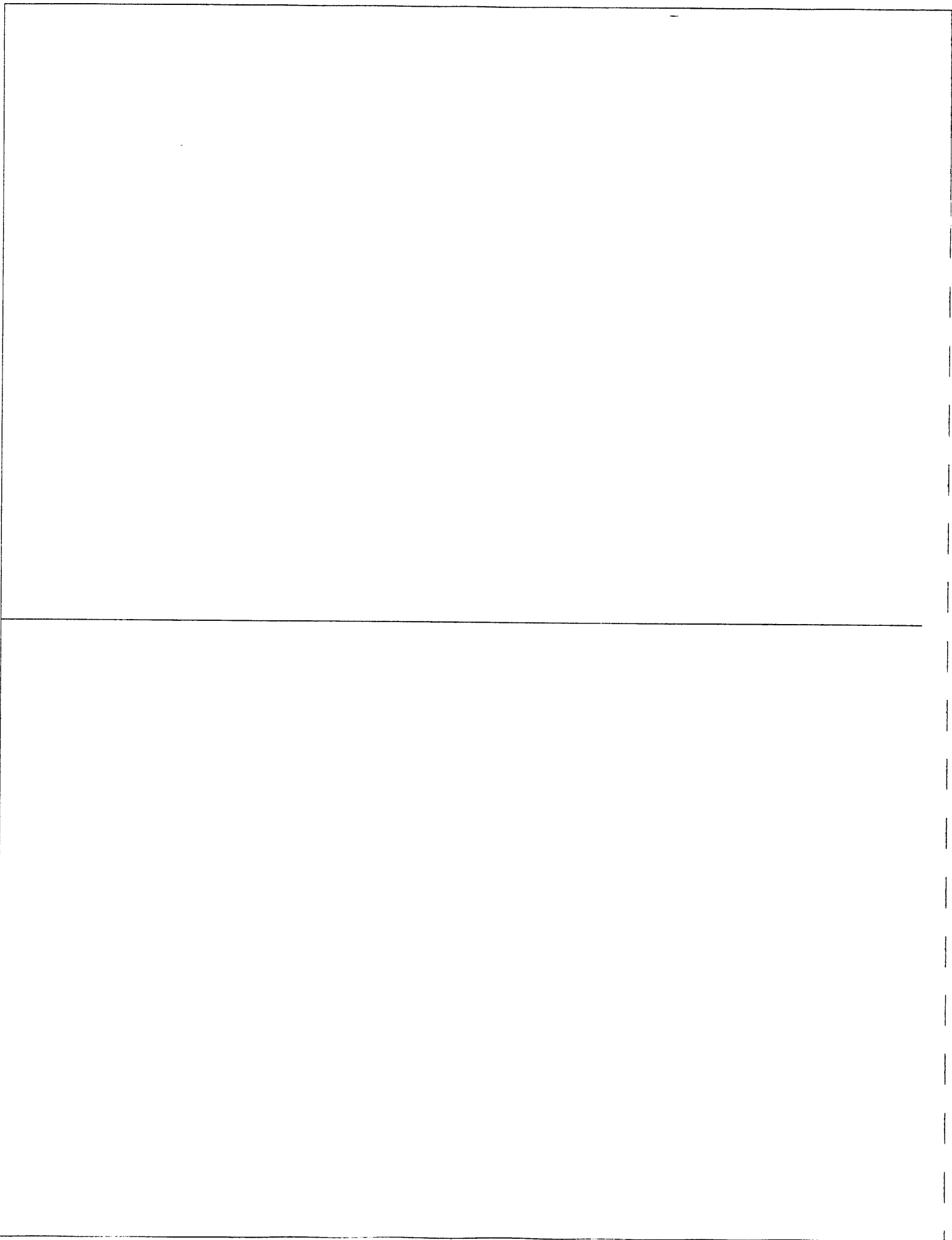
(The following is correct for 32bit abi programs)

Alias	Alternate Alias	Description
\$r0	\$zero	always 0
\$r1	\$at	reserved for assembler
\$r2..\$r3	\$v0..\$v1	expression evaluations, static links, returned values
\$r4..\$r7	\$a0..\$a3	arguments
\$r8..\$r15	\$t0..\$t7	temporaries
\$r16..\$r23	\$s0..\$s7	saved across procedure calls
\$r24..\$r25	\$t8..\$t9	temporaries
\$r26..\$r27	\$k0..\$k1	reserved for kernel
\$r28	\$gp	global pointer
\$r29	\$sp	stack pointer
\$r30	\$s8	saved across procedure calls
\$r31	\$ra	return address

New:

\$r4..\$r11

\$a0..\$a7



Module 9: Memory Management Overview

Memory Management Overview

9-1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Module Overview

This module provides an overview of the hardware and software mechanisms used to manage the system memory. Emphasis is on virtual addressing and memory paging, which are used to give users the illusion that their processes can consume all available memory or even more memory than physically available.

9-2

22jul1998

TR-IKI rev 0.7b SGI Proprietary

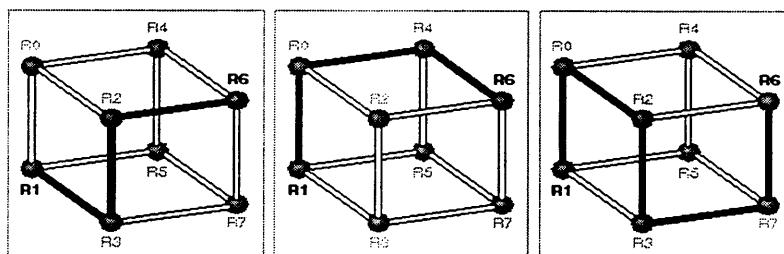
Module Objectives

After completing this module, you will be able to:

- Explain the characteristics of a swapping type of UNIX system.
- Describe the concepts of virtual address and memory page in IRIX systems.
- Describe the role of the TLB in memory address translation.
- Describe how virtual addresses are translated to physical memory addresses.
- Explain the concepts of demand paging and page stealing in IRIX.
- Use `sax(1)` to produce reports on memory, swapping, paging, and TLB activity.
- Use `ps(1)` to determine total size and current memory consumption of user processes.
- Use `gx_osview(1)` to display dynamic memory, swapping, paging, and TLB activity.

Hardware Memory Review

The hardware aspects of memory management were presented in the Hardware Overview section of this course. Please review those pages for the details. Following is a summary and diagram of the hardware memory concepts presented there.



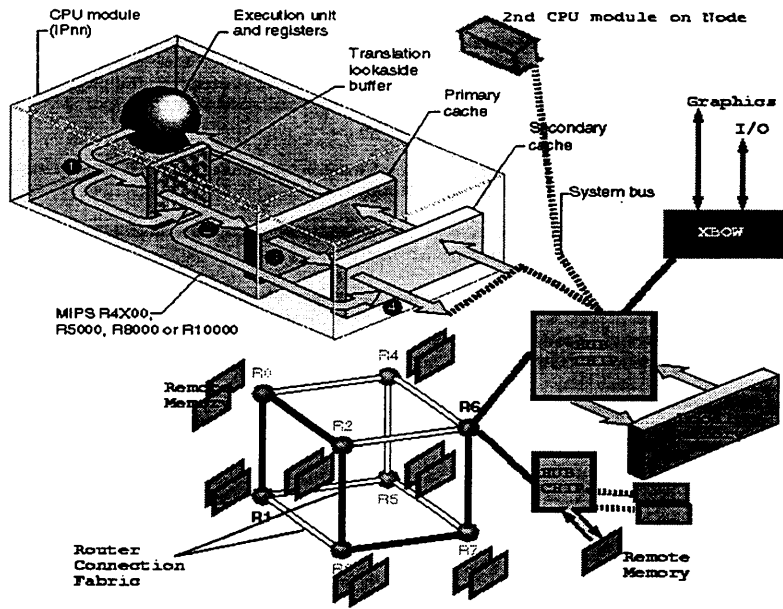
Origin2000 distributed-shared memory

- Located in a single shared address space but is physically dispersed across system nodes.
- Former systems had memory centrally located and only accessible over a single shared bus.
- The interconnection fabric is a mesh of multiple point-to-point links connected by the routing switches. These links and switches allow multiple memory accesses to occur simultaneously.
- To a processor, main memory appears as a single addressable space containing many blocks or pages.
- Page migration hardware moves data into memory closer to a processor that frequently uses it to reduce *memory latency*.

Origin2000 Memory Hierarchy (in order of increasing memory latency)

- Processor registers
- Cache (primary and secondary)
- Local memory
- Remote memory
- Remote caches

Hardware Address Sequence Review Diagram



Hardware Address Sequence Review Diagram Explanation

A CPU examines an instruction, and isolates that part of it which represents the address of the page, and the offset into that page, of the data that the CPU needs. This address might be something like the address of an instruction to fetch, or the address of an operand of an instruction. Then the CPU goes through the following steps in order to find that address.

1. The virtual address of the needed data is formed in the processor execution or instruction-fetch unit. Most addresses are then mapped from virtual to real through the Translation Lookaside Buffer (TLB). This process may have had a "TLB miss" if the virtual-to-physical mapping was not already in the TLB. At that point, the CPU had to exchange into kernel context in order to determine the physical address and then load it into the TLB. One way or another, at this point the TLB has a virtual-to-physical address mapping of the address the process wants, and the CPU 'knows' what physical page of memory it must access.
2. Most addresses are presented to the primary instruction or primary data caches, depending on what is being addressed. These caches are in the processor chip. If a copy of the data with that address is found, it is returned immediately.
3. When the primary cache does not contain the data, the address is presented to the secondary cache, which is used to hold both data and instructions. If the secondary cache contains a copy of the data, the data is returned immediately.
4. When the secondary cache does not contain the data, the physical address reference is placed on the system bus and handed over to the HUB chip. The HUB knows which areas of memory have been assigned to which nodes, which area of memory has been assigned as "local" to this node, and which nodes are attached to which router connections. The HUB acts as a switch, and directs the request either to this node chip's local memory, or whatever remote memory address is appropriate.
5. When the HUB chip recognizes that local memory does not contain the data, the address passes out through the "connection fabric", that is, through router connections to other nodes on this, or other hypercubes in the system, to a memory module in another node, from which the data is returned.

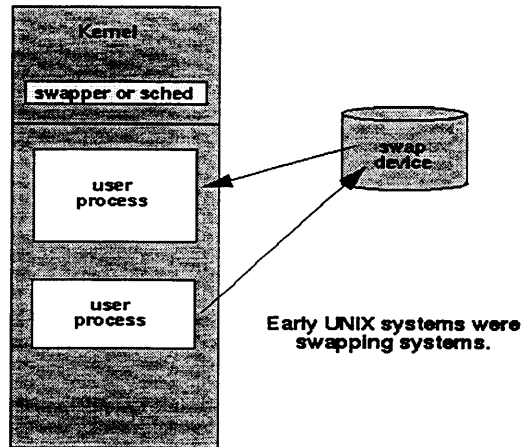
Memory Subsystem Introduction

One of the major concerns for the operating system is how it manages the finite amount of physical memory installed in the system hardware. The total amount of memory needed by all active processes on the system is constantly changing and generally is far greater than the actually available physical memory. The operating system kernel must handle situations like:

- Where will a process reside in main memory?
- How will it prioritize which processes are the most eligible to occupy main memory?
- What scheme will be used to move processes in and out of main memory?
- How will it allocate more memory to a process as its needs grow?
- How will it free unused memory when a process wants to shrink?

Historical Solutions to Memory Management (Swapping)

Earlier versions of UNIX used a method called *swapping* to manage main memory. With this method, whole processes were swapped from memory to disk to make room for other processes that needed to run, as shown below. Historically, UNIX was a swapping system and the swapping was done by a special process called the *swapper* or *sched* (short for *scheduler*) which always has a process ID (PID) of 0. More recent versions of UNIX still have a *swapper* process or *sched*.



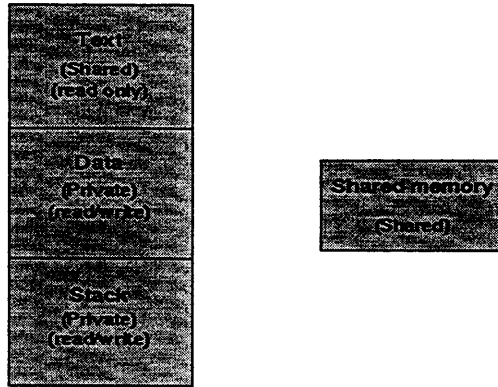
Recent Solution to Memory Management (Virtual Memory)

To overcome the constraints of having to have an entire process resident in physical memory in order to run, UNIX System V Release 4 adopted a concept referred to as *virtual memory*. A virtual machine allows programmers to ignore the physical layout and size of machine memory. A program is written to reference virtual addresses for both instructions and data, thus relieving the programmer from concern as to where things are physically located in memory.

Some attributes of virtual memory systems are:

- Gives illusion that there is more memory available than physically installed on machine.
- Can run programs that are larger than physical memory.
- Process does not have to be entirely in memory to execute.
- Translation mechanism is needed to convert virtual memory addresses to physical addresses at run time.

User Process Components Review



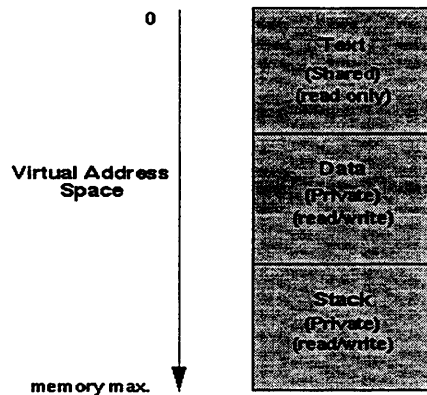
A *process* is the execution of a program and consists of a pattern of bytes that the CPU interprets as machine instructions (*text*), *data*, and *stack*. A program executing in a process reads and writes its data and stack areas and possibly shared memory areas. Following is a more complete description of the components that comprise a process.

- *Text* contains the executable code (machine instructions) for a process. It is usually marked read-only so that a process cannot alter its own code or be altered by other processes. Text areas can be shared by many user processes that are concurrently executing the same code; for example, multiple users using the same shell program (*sh*, *cs*, *ksh*).
- *Data* holds the data used and modified by the process during execution. It is usually marked for reading and writing. It is never shared with other processes; otherwise, a process could alter the data area of another process.
- *Stack* holds the data necessary for the program to call and return from code modules called subroutines and for allocation of local data

values. It is marked for reading and writing and cannot be shared with other processes.

- *Shared memory* is an area of memory accessible to multiple processes. One process can write data into the shared memory area and another process can read the data.

User Process Virtual Memory Image



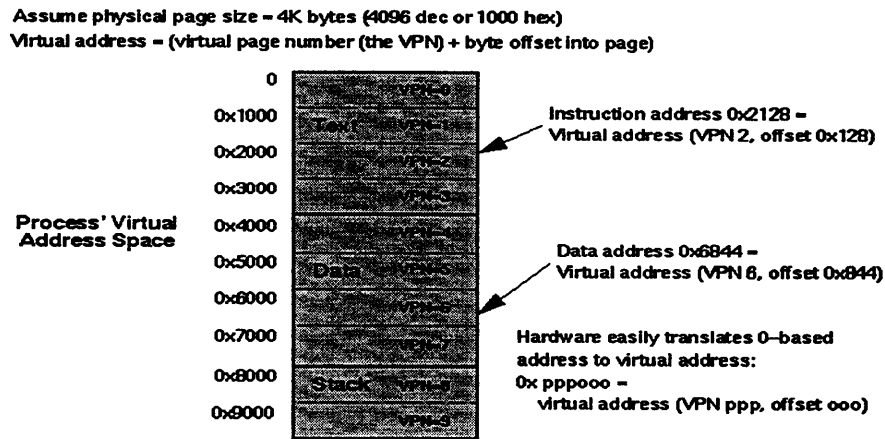
Assume that the physical memory of a system is addressable with the first byte located at byte offset 0, and that the last byte has a byte offset equal to the amount of memory on the system (in other words, the maximum physical byte memory location). Compilers (C, Fortran, C++, etc.) generate machine code that is 0-based, that is, the program is assumed to begin at byte offset 0 and consumes as many bytes as needed. If the system were to treat the compiler-generated addresses in a user's program as address locations in physical memory, it would be impossible to execute two processes concurrently because their addresses would overlap.

This is why compilers generate program addresses for a *virtual address space* within a given address range. The compiler assumes that every program begins at address 0 and can consume as much space as needed within the given range. The machine's memory management unit (MMU) then translates the virtual address generated by the compiler (0-based) into address locations in physical memory. The compiler does not need to know (nor does it care) where in physical memory the kernel will later load the program for execution. Furthermore, several copies of the same program can coexist in memory. They all would execute using the same virtual memory addresses but would be referencing different physical addresses.

User Process Virtual Addresses

Compilers generate *virtual addresses* (0-based) for data and instruction references without regard to the physical page size defined on the system. This makes program codes more portable from system to system. However, for purposes of this explanation of virtual addressing, assume that the machine's physical page size is defined to be 4K (4096 bytes). A process' virtual page will map onto a physical page somewhere in the system's memory (location controlled by the kernel) when the process executes.

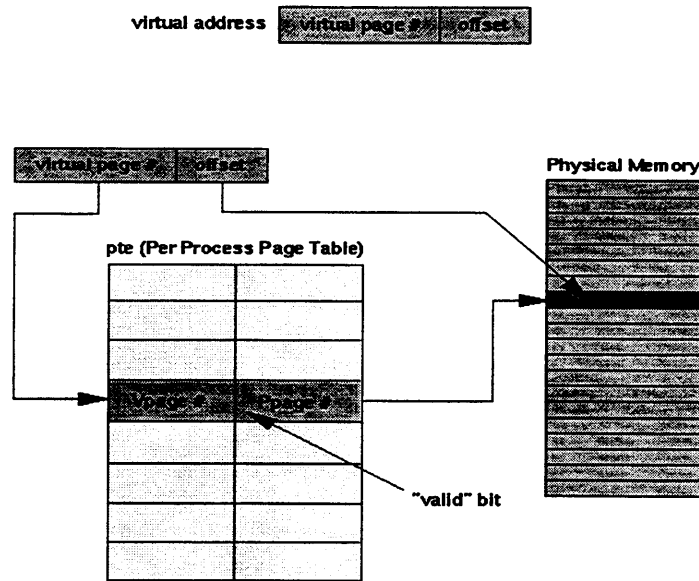
Every byte within a user's process is addressable with a virtual address. A *virtual address* consists of two parts: a virtual page number and an offset within that page.



The above illustration of a machine with physical page size defined as 4K bytes shows that the 0-based addresses generated by the compiler for instruction and data references actually serve as virtual addresses also. When the program executes, the CPU will interpret the 0-based addresses as virtual addresses and map the virtual addresses to physical memory locations.

It is easy for the CPU to translate a compiler-generated (0-based) address into a virtual address. For a system with page size defined as 4K bytes (as above), the rightmost 12 bits in the address (remember 1 hex digit = 4 binary digits) are the byte offset into the page and the remaining leftmost bits comprise the virtual page number (VPN). Likewise, for a system with page sizes of 16K bytes, like the Cray Origin2000, the rightmost 14 bits in the address are interpreted as the byte offset.

Virtual to Physical Address Translation

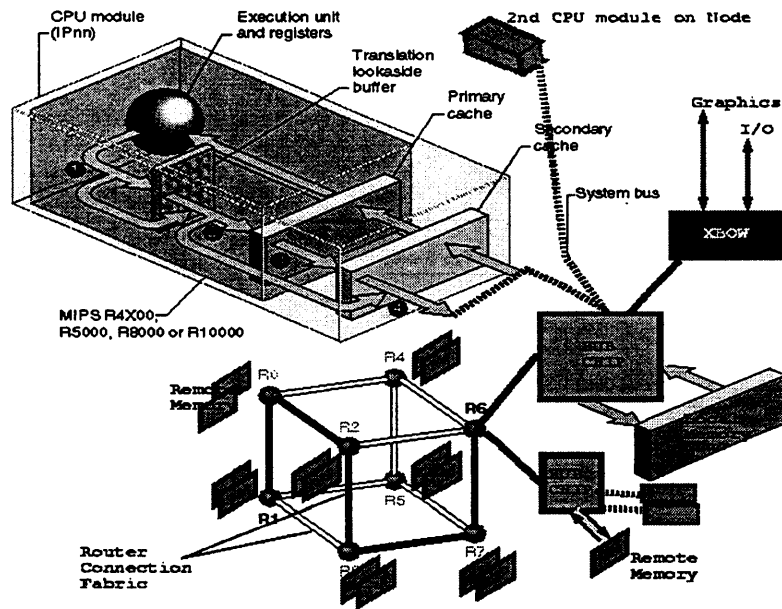


As a user process executes, it references instruction code and data by using the virtual addresses generated by the compiler. These virtual addresses are transparently translated into physical addresses by a combination of hardware and software.

Every process has its own page table which provides a mapping from the process' virtual address space (virtual page numbers) to physical memory locations (physical page numbers). Using a combination of hardware and software, a process's virtual page number is looked up in its pte (page table) to produce a physical page number. The physical page number is then combined with the page offset to yield a real address in physical memory.

A process's memory space does not have to be entirely resident in physical memory at once. Only the pages currently being referenced need to be memory resident. Therefore, virtual addressing allows a process' virtual address space to be larger than the machine's physical address space. The kernel keeps track of which pages are currently in memory by maintaining a flag in each page table entry (called the *valid* bit). If a page is not currently in main memory then it is invalid and the memory management system must keep information about where the page is residing in secondary storage. When a process references a non-resident page (invalid), then the process must wait until the system brings the page into physical memory.

Translation Lookaside Buffer (TLB)



The pte (per-process page table) is a structure maintained in physical memory. Each time a process references memory, the virtual page number needs to be looked up in the process's page table to locate the physical page of memory where that virtual page is mapped. However,

9-17

22jul1998

TR-IKI rev 0.7b SGI Proprietary

searching the page table every time a process references memory would be very damaging to the process' performance. Therefore, if a process's page table (or portion of it) could be stored in memory built into the CPU's chip, then virtual to physical address translations could be performed very quickly. This type of memory is referred to as *associative memory*.

The purpose of associative memory is to "associate" a given virtual page number to a physical page number. However, associative memory on the CPU is limited to a small area due to the lack of space for this type of memory on the chip. On MIPS CPUs, this area is called the Translation Lookaside Buffer (TLB). The number of TLB entries varies by MIPS processor type.

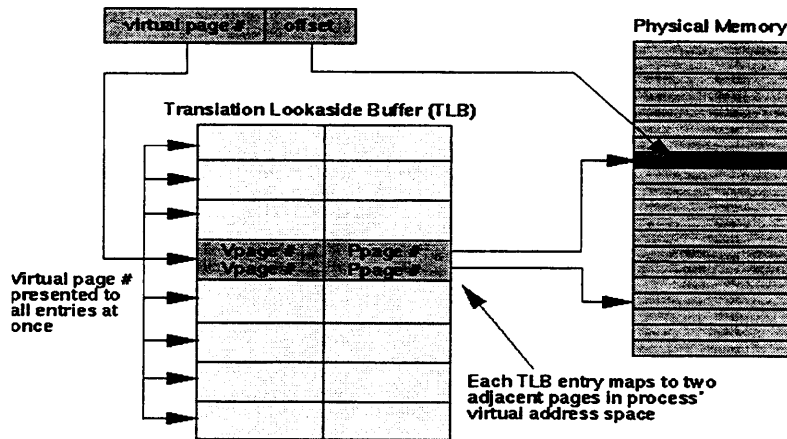
Processor Type	Number of TLB Entries
R4x00	96
R5000	96
R8000	384
R10000	128

9-17.a

22jul1998

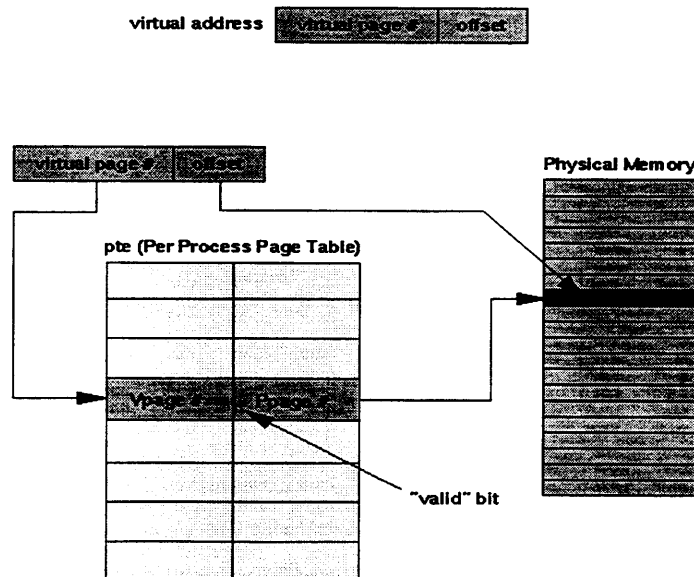
TR-IKI rev 0.7b SGI Proprietary

Translation Lookaside Buffer (TLB) (continued)



When a user's virtual address is presented to the CPU, the TLB is first checked for a match on the virtual page number. The virtual page number is presented to all TLB entries at the same time. Note that each TLB entry points to two adjacent pages within the process' address space.

TLB "Hits" and "Misses"

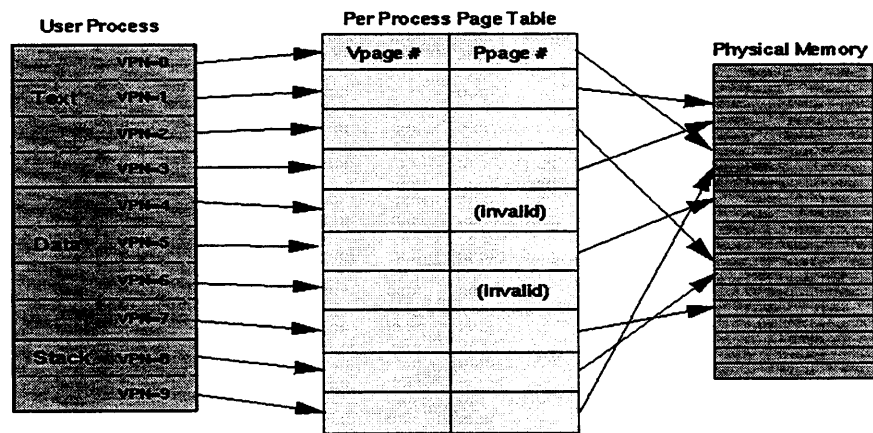


Ideally, the TLB would be large enough to hold an entry to translate every possible virtual address presented to the CPU by a process during its execution. However, large TLBs are not practical and they can only hold a subset of the page table entries for a given process. Each TLB entry can map to two adjacent pages in the user process' virtual address space. These pages do not need to be adjacent in physical memory.

When the virtual memory address requested in the CPU falls within a page described by a TLB entry, the TLB supplies the physical memory address for the desired page. The offset is then applied to locate any desired byte location in physical memory. This is referred to as a TLB hit.

When the virtual memory address requested in the CPU is not covered by any active TLB entry, the MIPS processor generates an interrupt to the kernel which is then handled by an IRIX kernel routine. The kernel inspects the requested address. If the address is found to be valid (in other words, resides within the process' virtual address space), the kernel loads a TLB entry from the appropriate entry in the process' page table. The kernel then restarts the instruction which now will find an appropriate TLB entry to perform the address translation.

Virtual Addressing Summary



This illustration summarizes how virtual addresses within a user's process are translated to locations within physical memory. At compile time, a user's program is compiled using 0-based addresses to locate instructions and data. These 0-based addresses are referred to as virtual addresses and consist of two parts: the virtual page number and a byte offset within the page. At execution time, when these virtual addresses are presented to the CPU for resolution, they must be translated to physical addresses in real memory.

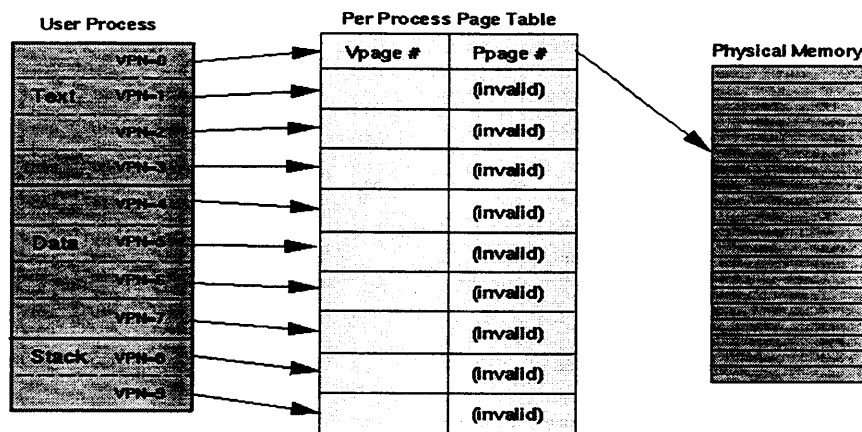
When a process is loaded into memory to execute, the kernel establishes a pte (page table) for the process. Each virtual page within the process' virtual address space will have an entry in the page table. If a particular page currently resides in physical memory, the page table will point to where it is located in real memory. Otherwise, the virtual page is marked "invalid" in the page table.

The task of translating virtual addresses occurs in the TLB. The TLB is an on-chip associative memory limited in size. The TLB basically

contains a subset of the entries in the process' page table. If the CPU finds a match on the desired virtual page in the TLB, this is considered a "TLB hit". It is quick and easy to determine the page's physical address in memory. If the CPU does not find a match on the desired virtual page in the TLB, this is a "TLB miss". The kernel must get involved to load a TLB entry with the appropriate entry from the process' page table and then re-issue the affected instruction.

The physical pages that correspond to a user's process can be anywhere within the user portion of system memory. When the kernel assigns physical pages of memory to a process, it need not assign the pages contiguously or in any particular order. The purpose of paged memory is to allow greater flexibility in assigning physical memory.

Demand Paging Overview



When a process is created on the system, only a small amount of physical memory is initially consumed. Some memory is needed by the kernel to control and manage each process. The code (text) and data areas associated with the new process remain in the file containing the program which is being executed. Therefore, most of the page table entries for a newly created process would be marked invalid.

Pages are created and allocated for a process only when they are referenced by the currently running process. This mechanism is referred to as *demand paging*. The entire process does not need to reside in memory in order to execute. The kernel loads pages of a process on demand when the process references the pages.

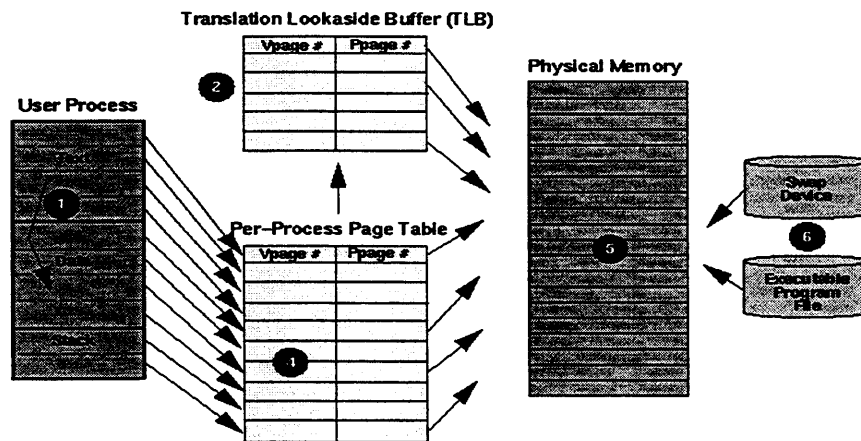
With demand paging, physical memory pages are created for only the parts of the program which actually execute. The parts of a program that never execute can remain in secondary storage. An example of a piece of program that might not execute would be error or signal

handling code which will not execute unless there is an error or signal is delivered.

Processes tend to execute instructions in smaller portions of their entire text (instruction) space, such as in looping constructs and frequently called subroutines. Also, a process tends to reference data in small subsets or clusters of the process' total data space. Each process has a set of pages that need to be in main memory to ensure it runs efficiently. This set of pages is referred to as its *working set*. As a process executes, its working set changes depending on its pattern of memory references.

When a process tries to access an address that is not in the working set, a *page fault* occurs so that the kernel can read into memory the page containing the desired address and attach the page to the process's address space. The kernel suspends the execution of the user process until it has read the needed page into memory and attached it to the process's address space. After the page has been loaded into memory, the process re-issues the instruction it was executing when it incurred the fault.

Demand Paging Page Load Procedure



The procedure for loading a page into memory is as follows:

1. The process references a memory address.
2. The CPU attempts to translate the user's virtual address in the TLB. Assume no entry in the TLB can translate the virtual address to physical address.
3. The currently running process is suspended and a fault is generated. This is called a *TLB miss* and control passes to the IRIX kernel.
4. The kernel's exception handler searches the process's page table for a valid entry corresponding to the virtual address that was not

found in the TLB. If one is found, the entry is placed in the TLB and the instruction is re-issued. Now when the instruction is executed, the virtual address is found in the TLB (TLB hit), translated to the physical memory address, and accessed by the CPU.

5. If the kernel's exception handler cannot find a matching entry in the page table, then the desired page is not residing in physical memory. The kernel then locates a free page in physical memory and associates it with the current executing process by adding an entry to the process' page table.
6. The kernel then initiates an input operation to fetch the requested data from either the file system where the executable program resides or the swap device. The process then voluntarily goes to sleep allowing other processes to run while the input operation is in progress. After the data arrives in memory, the process is 'awakened and again scheduled to run.

Demand Paging Advantages and Disadvantages

The IRIX kernel supports a demand paging algorithm which means that pages of memory are swapped between main memory and a swap device. This kernel feature gives the illusion that a single user process has all of system memory available if needed.

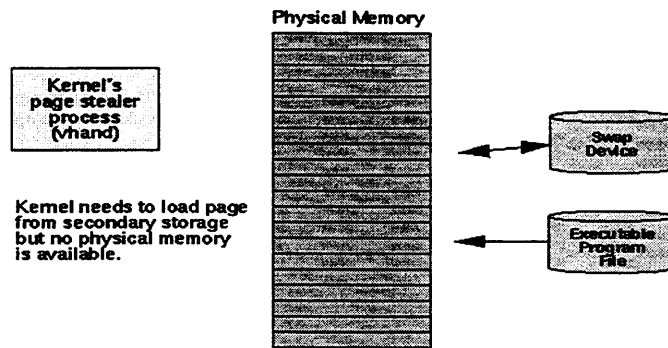
Some advantages of demand paging systems:

- Frees processes from size limitations otherwise imposed by the amount of physical memory available on the system.
- Transparent to user programs.
- Allows more processes to fit simultaneously into main memory as compared to a swapping system.

Some disadvantages of demand paging systems:

- Processes must wait for a page while it is being loaded.
- During initial stages of a process, a process will usually generate many page faults which leads to slower startup times and many disk operations.

Page Stealing



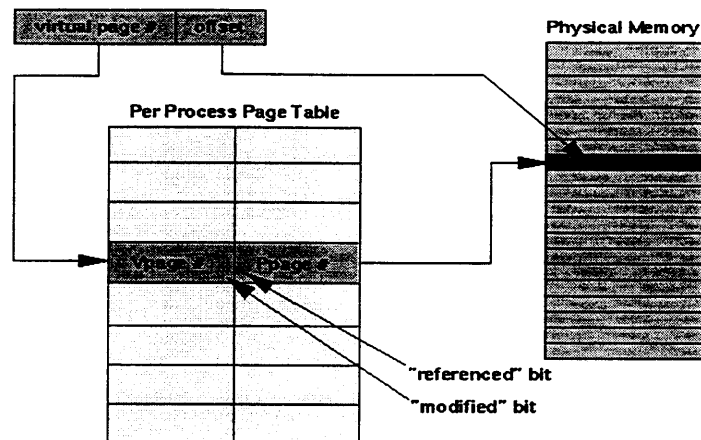
Eventually the operating system will need to bring into memory a page from secondary storage (swap device or executable program file) but all physical memory pages are in use. The kernel handles this situation by a mechanism called *page stealing*.

The kernel has a process called the *page stealer* (or *vhand*) that swaps out memory pages to the swap device. The kernel creates the page stealer during system initialization and invokes it throughout the lifetime of the system whenever the system is low on free pages; in other words, whenever the number of free pages falls below a configurable threshold (called the low-water mark).

The page stealer examines pages that are already allocated to a process and steals some of them so that they can be used by other processes. It keeps stealing pages from processes until the number of free pages reaches a configurable threshold (called the high-water mark).

Note that the low-water and high-water thresholds need to be set appropriately in order to reduce the frequency that the page stealer needs to execute. Otherwise, the page stealer process can get into a thrashing situation where it is being called very frequently with little work to do; thus, negatively impacting system performance.

Page Stealing Page Selection



The page stealer has to decide which pages are the best candidates to steal. The best candidates are those pages whose next reference will be the farthest into the future. Since that is very difficult to predict, the most common method used in UNIX System V systems is called *Not Recently Used* (NRU).

With the *Not Recently Used* method, every page has a *modified* and *referenced* bit in its page table entry. When a page is referenced, its referenced bit is set. Likewise, when a page is modified, its modified bit is set.

When the page stealer needs to steal some pages, it does so in the following order:

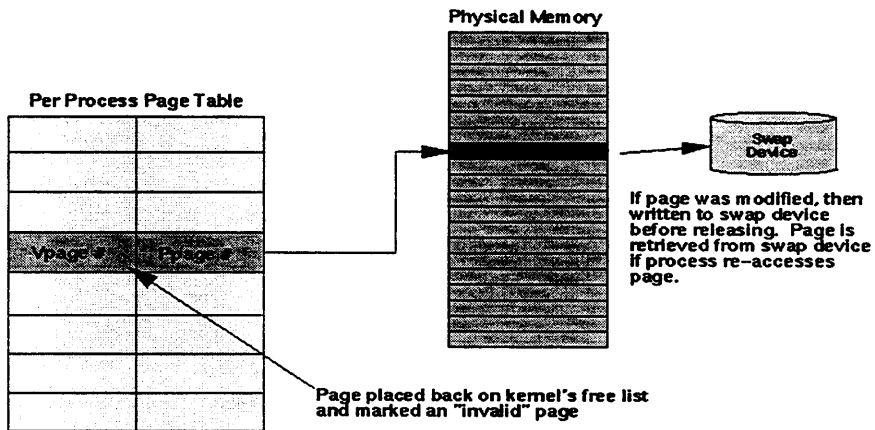
- First selects those pages which have not been referenced for "a long time". Pages that are not included within a process's working set

are ideal candidates.

- If that does not yield enough pages, then it selects those which have been referenced but not yet modified.
- If that still does not yield enough pages, then it selects those that have been referenced and modified.

Eventually, most of the pages will have their referenced bit set, so the page stealer makes a sweep through memory clearing the referenced bit of every page in memory.

Page Stealing Page Actions



After selecting which pages will be stolen, the following actions are taken:

- If page has not been modified:

Page is simply placed back on the list of free pages. These pages are invalidated in their respective page tables and cleared from any TLB entries pointing to them. Future access to these pages will require reloading from the swap device or secondary file storage. However, stolen pages are added to the back of the kernel's free list of pages so that they can be quickly reclaimed by the original owning process.

- If page has been modified (*dirty* page):

9-26

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Page is first written to the swap device before being placed on the list of free pages.

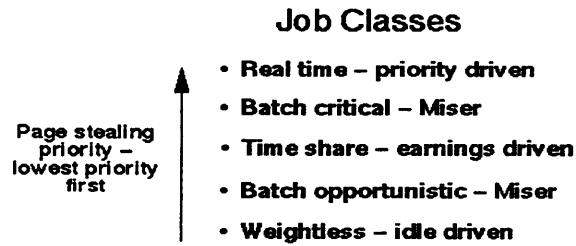
Note that under heavy load conditions, it is possible for a process to have less than its working set of pages available in main memory. This condition can lead to excessive kernel paging because immediately after a page is stolen from a process, the process may need it to be paged back in. This thrashing situation may be going on in every active process on the system, thus causing excess system overhead because much of the system resources are being devoted to paging in and paging out of processes instead of getting user work done. If a system is thrashing, entire processes may have their pages stolen and written out to the swap device. In some cases, large processes will be killed to free up memory (see "The Swapper Process in IRIX", below). If a system is constantly swapping processes in and out, this may be evidence that the system does not have enough installed physical memory.

9-26.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Page Stealing and Job Classes



Work is managed in an IRIX system within job classifications. When the page stealer needs to release memory pages, it will first apply its page selection criteria (discussed above) to processes representing jobs in the lowest classifications first, and then work its way up through the above ordered list.

The IRIX job classifications are listed above with brief explanations below:

- **Weightless**

A job that is about to go idle is placed in this class.

- **Batch opportunistic**

Batch requests submitted to Miser are placed in this class and are specified with CPU time requirements. Job will complete when there is opportunity. If job cannot complete in specified time interval, it is moved to the Batch critical class.

- **Time share**

Typical interactive IRIX processes placed in this class.

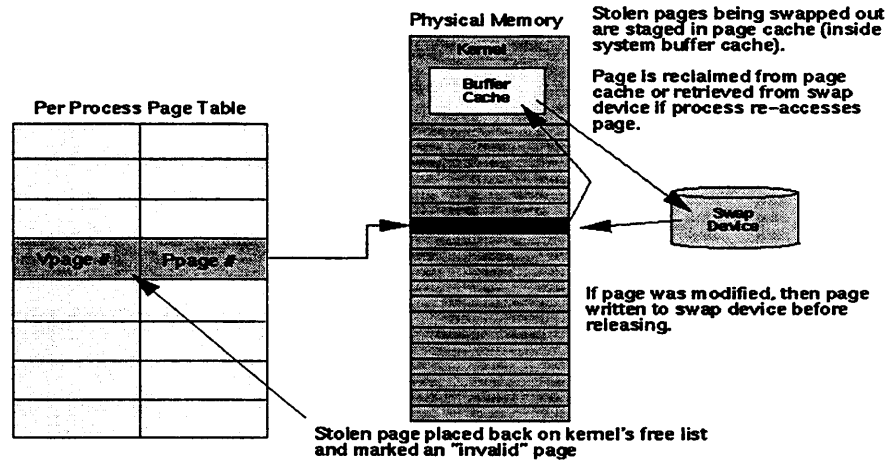
- **Batch critical**

Batch requests are submitted to Miser with a specified CPU time limit and placed into the batch opportunistic class. If job cannot complete in specified interval of time, job moves to this class with higher priority.

- **Real time**

Highest priority jobs in the system. Guaranteed a specified amount of CPU attention in specified interval.

Page Cache in IRIX



When IRIX steals a page from a process and that page has been modified by the process, it cannot be released but must be written to the swap device. IRIX implements an intermediate staging area for those pages which are being moved to the swap device. This area is called *page cache* and resides within the kernel's buffer cache. The kernel normally uses the buffer cache for an intermediate staging area for I/O operations. Modified stolen pages are temporarily staged in the page cache before being written to the swap device. The kernel assumes the overhead of writing the stolen pages to the swap device later when they age out of the buffer cache.

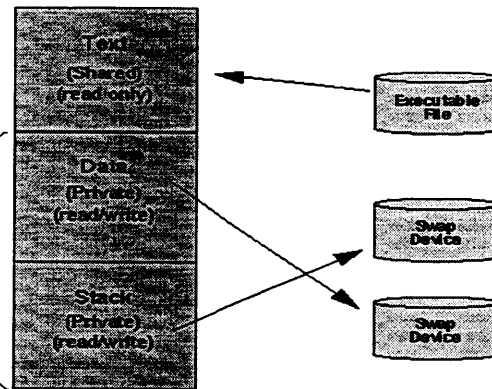
The page cache allows the *page stealer* greater performance because it does not have to wait for completion of physical I/O to the swap device. Also, if a process re-accesses the stolen page while it still resides in the page cache and before it has actually been written to the swap device, it can be reclaimed from the page cache simply and efficiently.

User Process Space and Swapping

Text pages are read-only. Stolen text pages are not swapped, but simply released and re-read from the executable file when accessed again.

Anonymous memory pages – not associated with executable file.

Data and stack regions are writeable and pages are generated when needed by IRIX (anonymous). Stolen data and stack pages are swapped to a swap device.



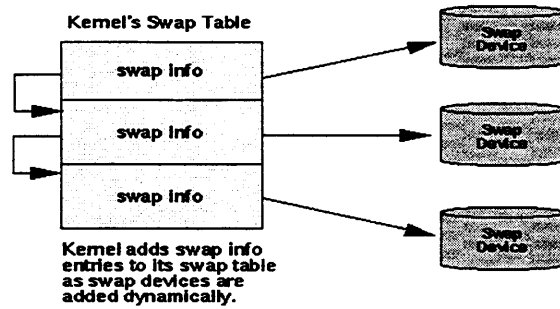
A process's memory space is partitioned into several regions. The user is able to modify the memory in some of these regions but not others.

The text region has read-only status and is shareable with other processes. Text regions are not allowed to be modified. This means that if the page stealer stole a text page from a process, if that page is needed again by the process, that page can simply be reloaded from the executable file image (because the text page has not been modified). This also means that the page stealer can simply release the text page and no swapping to a swap device is necessary.

Conversely, the data and stack regions have read/write status and can be modified. The IRIX kernel allocates pages for the data and stack regions as needed by the process. The pages allocated for the data and stack areas are *not* associated with the executable file from which the user program was loaded. Therefore, these pages are referred to as *anonymous* memory pages associated with the process. If the page stealer

needs to steal an anonymous page, then it must find a location on secondary storage where it can temporarily store the page for later recall, if accessed. The swap device(s) serve this role for the page stealing operation.

Swap Space Management

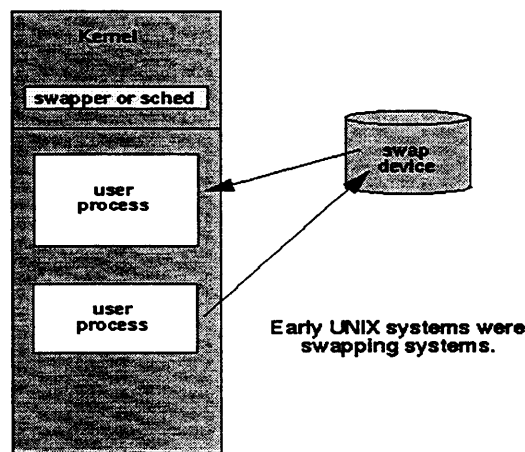


As explained on the previous page, only anonymous memory pages associated with user processes need to be swapped with the kernel's page stealing operation. The IRIX kernel then needs to maintain a mapping between anonymous pages and the swap space. Each anonymous page is mapped to a page-sized block of swap space.

IRIX must have at least one disk partition or file allocated for swap space. Additional swap areas can be added or removed while the system is running. When a swap area is added by the system administrator, the number of pages that can be stored in that swap area is calculated. The kernel then adds a *swap info* structure to its swap table to keep track of the new swap area.

When new anonymous memory pages are generated by user processes, the kernel's swap management routines spread the anonymous pages across all swap areas to maintain performance. If some swap areas are full, all swap areas will be searched until free space is found.

The Swapper Process



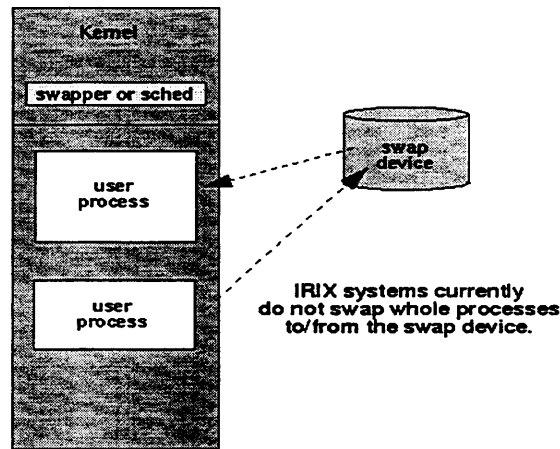
As stated at the beginning of this section, earlier versions of UNIX used a method called *swapping* to manage main memory. With this method, whole processes were swapped from memory to disk to make room for other processes that needed to run. Historically, UNIX was a swapping system and the swapping was done by a special process called the *swapper* or *sched* (short for scheduler) which always has a `PID=0`. More recent versions of UNIX still have a *swapper* process or *sched*.

The reason that demand paging type systems still need a *swapper* process is because there can be times when demands for memory are so high that the page stealer cannot maintain a large enough list of free pages. When free memory falls below a specified level, the kernel's *swapper* or *sched* process is invoked. The *swapper* process calls a kernel function to select a process to swap out to the swap device. All of the memory pages associated with the selected process are then freed. A flag is cleared in the selected process' `proc` table entry indicating it

is no longer eligible to run.

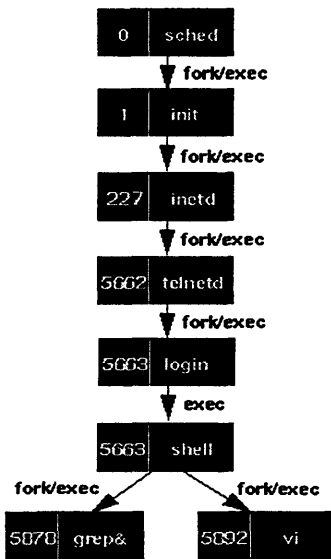
At a later time, the kernel's *swapper* or *sched* process is invoked again. If the amount of free memory is above a specified level, a kernel function is called to select a process to swap back into memory (now residing on the swap device). A flag is set in the selected process' *proc* table entry indicating it is now eligible to run again. As this process receives CPU attention, the process's memory pages will be faulted back into main memory when they are accessed, by the demand paging mechanism described earlier in this section.

The Swapper Process in IRIX



The implementation of *sched* or the *swapper* process in IRIX systems is different than a typical UNIX system. The *swapper* process is implemented such that it never swaps whole processes to and from a swap device. Currently, if memory is oversubscribed to an extent where the page stealer cannot keep up with the demand, then IRIX will begin removing processes from the system. The largest processes are the first candidates.

The Swapper Process Relationship to Other Processes



The *swapper* or *sched* process is a special kernel process which serves as the origin (or great-great-... grandparent) of all processes on the system. For instance, the *swapper* generates the *init* process which is responsible for initiating all of the major daemons that run on the system. The reason *sched* always has a process ID (PID) value of zero is because it is always the first process created on the system.

The diagram to the left shows how the *swapper* or *sched* is related to all other processes on the system.

Reporting Paging Activity (sar -p)

The *sar(1)* command (with option *-p*) will report paging activity for the host system.

```

$ sar -p
IRIX64 flurry 6.5-ALPHA-1274427934 02121253 IP27 02/27/98
05:55:50 vflt/s dfill/s cache/s pgswp/s pgfil/s pflt/s cpyw/s steal/s rclm/s
05:55:50 unix restarts
06:00:06 152.97 32.16 117.73 0.00 0.68 177.21 105.60 91.13 0.03
06:10:06 10.60 2.14 8.46 0.00 0.33 11.74 6.46 6.21 0.00
06:20:06 10.74 1.87 8.87 0.00 0.00 12.70 8.10 5.81 0.00
06:30:06 6.08 1.17 4.91 0.00 0.00 7.59 4.21 3.79 0.00
06:40:07 2083.65 569.09 1503.61 0.00 0.83 1201.67 631.85 874.68 0.00
06:50:06 4682.61 1312.76 3351.06 0.00 1.68 2760.90 1462.15 2022.61 0.00
07:00:07 5056.14 1221.84 3814.96 0.00 0.16 3099.43 1698.52 2051.97 0.00
07:10:06 4752.46 1327.61 3389.85 0.00 0.05 2812.26 1470.38 2052.98 0.00
07:20:06 3994.62 1335.02 2619.04 0.00 0.06 2323.55 1137.50 1915.22 0.00
07:30:06 3945.11 1260.85 2628.23 0.00 0.32 2265.80 1119.86 1853.64 0.00
07:40:06 3488.68 1144.82 2309.96 0.00 0.13 1983.08 993.95 1641.42 0.00
07:50:06 3465.20 962.82 2472.63 0.00 0.26 2124.83 1126.90 1543.53 0.00
08:00:06 4118.88 1034.86 3061.75 0.00 0.21 2607.35 1420.63 1763.54 0.00
(abbreviated)
14:00:07 346.31 199.52 141.92 0.00 13.94 79.14 30.21 212.16 0.00
14:10:06 290.03 153.21 130.94 0.00 15.02 92.87 47.74 176.72 0.00
14:22:30 unix restarts
14:30:06 2009.49 323.00 1682.50 0.00 133.46 1691.78 92.25 1347.28 0.02
14:40:06 3142.42 182.61 2952.53 0.00 272.73 2887.67 57.58 1899.82 0.00
14:50:06 385.55 233.25 150.89 0.00 5.17 112.84 58.06 273.47 0.00
15:00:07 194.89 29.92 157.64 0.00 7.80 96.63 29.94 71.55 0.00
15:10:06 56.39 31.62 24.77 0.00 0.34 35.87 17.19 42.64 0.00
15:20:06 101.87 73.30 28.41 0.00 0.00 49.48 21.50 86.59 0.00
Average 1637.73 456.06 1171.52 0.00 15.99 974.88 435.29 747.94 0.00
$
  
```

The sar(1) output data columns have the following interpretation (/s means per second):

Column header	Interpretation
vflt/s	Address translation page faults (valid page not in memory)
dfill/s	Address translation fault on demand fill or demand zero page
cache/s	Address translation fault page reclaimed from page cache
pgswp/s	Address translation fault page reclaimed from swap space
pgfil/s	Address translation fault page reclaimed from file system
pflt/s	(Hardware) Protection faults -- including illegal access to page and writes to (software) writable pages
cpyw/s	Protection fault on shared copy-on-write page
steal/s	Protection fault on unshared writable page
rclm/s	Pages reclaimed by paging daemon

Reporting System Swapping and Switching Activity (sar -w)

The sar(1) command (with option -w) will report system swapping and switching activity for the host system.

```

$ sar -w
IRIX64 flurry 6.5-ALPHA-1274427934 02121253 IP27 03/01/98
01:37:05 swpin/s bswin/s swpot/s bswot/s psvot/s pswch/s kswch/s
01:37:05          unix restarts
01:40:06 0.00 0.0 0.00 0.0 0.00 181 632
01:50:06 0.00 0.0 0.00 0.0 0.00 20 557
02:00:07 0.00 0.0 0.00 0.0 0.00 30 599
02:10:07 0.00 0.0 0.00 0.0 0.00 822 575
02:20:07 0.00 0.0 0.00 0.0 0.00 24 566
02:30:06 0.00 0.0 0.00 0.0 0.00 23 563
02:40:06 0.00 0.0 0.00 0.0 0.00 30 564
02:50:06 0.00 0.0 0.00 0.0 0.00 22 560
03:00:07 0.00 0.0 0.00 0.0 0.00 21 558
03:10:06 0.00 0.0 0.00 0.0 0.00 20 558
03:20:06 0.00 0.0 0.00 0.0 0.00 22 560
03:30:06 0.00 0.0 0.00 0.0 0.00 25 563
03:40:06 0.00 0.0 0.00 0.0 0.00 21 559

          (abbreviated)

07:20:06 0.00 0.0 0.00 0.0 0.00 456 601
07:30:06 0.00 0.0 0.00 0.0 0.00 694 699
07:40:07 0.00 0.0 0.00 0.0 0.00 634 772
07:50:06 0.00 0.0 0.00 0.0 0.00 624 813
08:00:07 0.00 0.0 0.00 0.0 0.00 607 747
08:10:06 0.00 0.0 0.00 0.0 0.00 924 850
08:20:06 0.00 0.0 0.00 0.0 0.00 1199 1002
Average 0.00 0.0 0.00 0.0 0.0 187 610
$

```

The `sar(1)` output data columns have the following interpretation (/s means per second):

Column header	Interpretation
swpin/s	Pages swapped in
bswin/s	Number of 512-byte units swapped in
swpot/s	Pages swapped out
bswot/s	Number of 512-byte units swapped out
pswot/s	Processes swapped out
pswch/s	Processes switched
kswch/s	Kernel switches

Reporting TLB Activity (`sar -t`)

The `sar(1)` command (with option `-t`) will report TLB activity for the host system.

```
$ sar -t
IRIX64 flurry 6.5-ALPHA-1274427935 02241507 IP27 03/02/98
08:50:06 tflt/s rflt/s vawrp/s sync/s flush/s idwrp/s idget/s idprg/s vawprg/s
09:00:06 0.00 0.20 0.00 2.76 352.82 0.00 220.70 946.84 0.00
09:10:06 0.00 3.45 0.00 51.19 6552.07 0.00 464.64 1854.08 2.55
09:20:06 0.00 0.93 0.00 10.94 1400.16 0.00 470.48 2057.31 0.02
09:30:06 0.00 2.89 0.00 38.76 4960.34 0.00 511.25 2124.62 1.68
09:40:06 0.00 0.73 0.00 9.98 1277.49 0.00 363.50 1580.55 0.00
09:50:06 0.00 0.59 0.00 8.12 1039.60 0.00 351.25 1432.23 0.07
10:00:06 0.00 0.79 0.00 9.02 1155.18 0.00 367.74 1476.70 0.04
Average 0.00 1.37 0.00 18.69 2392.05 0.00 392.83 1639.04 0.62
$
```

The `sar(1)` output data columns have the following interpretation (/s means per second):

Column header	Interpretation
tflt/s	User page table or kernel virtual address translation faults: address translation not resident in TLB
rflt/s	Page reference faults (valid page in memory, but hardware valid bit disabled to emulate hardware reference bit)
sync/s	TLBs flushes on all processors
vmwrp/s	Syncs caused by clean (with respect to TLB) kernel virtual memory depletion
flush/s	Single processor TLB flushes
idwrp/s	Flushes because TLB ids have been depleted
idget/s	New TLB ids issued
idprg/s	TLB ids purged from process
vmprg/s	Individual TLB entries purged

Process Size (`ps -l`)

The `ps(1)` command (specifying the `-l` option) will display the total size of individual processes as well as the amount of main memory currently being consumed by those processes. The example below shows a typical `ps` display. Sizes are listed in units of pages of memory.

```

$ ps -lf
  F S      UID   PID  PPID  C  PRI  NI   P   SZ:RSS      WCHAN      STIME TTY      TIME CMD
b0 S      hlm   9712  8849  0  64  24  *   902:0      883b8310    Feb 11 ttyq0  0:01 xcalc
b0 R      hlm   22726 8849  8  64  20  0   406:169    - 10:07:44 ttyq0  0:00 ps -lf
b0 S      hlm   8849  8848  0  60  20  *   102:60     882b3754    Feb 05 ttyq0  0:05 -ksh
$

```

Total size of process ▲ ▲ Total resident size of process

The SZ and RSS columns of the `ps(1)` display are explained on the `ps(1)` man page and reproduced below:

- SZ** Total size (in pages) of the process, including code, data, shared memory, mapped files, shared libraries and stack. Pages associated with mapped devices are not counted. (Refer to `sysconf(1)` or `sysconf(3C)` for information on determining the page size.)
- RSS** Total resident size (in pages) of the process. This includes only those pages of the process that are physically resident in memory. Mapped devices (such as graphics) are not included. Shared memory (`shmget(2)`) and the shared parts of a forked child (code, shared objects, and files mapped `MAP_SHARED`) have the number of pages prorated by the number of processes sharing the page. Two independent processes that use the same shared objects and/or the same code each count all valid resident pages as part of their own resident size. The page size can either be 4096 or 16384 bytes as determined by the return value of the `getpagesize(2)` system call.

Reporting Memory Statistics (sar -R)

The sar(1) command (with option -R) will report memory statistics for the host system.

```

$ sar -R
IRIX64 flurry 6.5-ALPHA-1274427934 02121253 IP27 02/27/98
05:55:50 physmem kernel user fsctl fsdelwr fsdata freedat empty
05:55:50          unix restarts
06:00:06 2342912 38172 2103 398 23 4505 346 2297365
06:10:06 2342912 38354 1800 431 10 4940 340 2297037
06:20:06 2342912 38379 1761 481 10 4957 340 2296984
06:30:06 2342912 38400 1784 483 11 4997 339 2296898
06:40:07 2342912 39282 2777 1011 93 7225 1342 2291182
06:50:06 2342912 40060 6397 1089 118 7576 2134 2285538
07:00:07 2342912 40404 3220 1172 327 8661 2832 2286296
07:10:06 2342912 40508 3288 1207 152 8445 3566 2285746
07:20:06 2342912 40724 8382 1261 148 8667 3614 2280116
07:30:06 2342912 40982 5245 1332 8002 12420 3708 2271223
07:40:06 2342912 41110 18805 1364 163 23415 4637 2253418
07:50:06 2342912 41444 39508 1428 134 242072 21059 1997267
08:00:06 2342912 41802 41616 1594 746 247114 16813 1993227
08:10:06 2342912 41931 39533 1646 87 245380 22098 1992237
08:20:06 2342912 42365 42821 1731 235 252147 23317 1980296
08:30:06 2342912 43025 43530 1946 113 278520 24120 1951658
08:40:07 2342912 43223 40950 2002 171 272597 24796 1959173
08:50:07 2342912 43508 40791 2166 68 274590 25790 1955999
09:00:06 2342912 43602 46226 2197 22 271718 27683 1951464

          (abbreviated)

13:50:06 2342912 52736 186004 2337 941 462051 244577 1394266
14:00:07 2342912 52900 187836 2342 550 465263 262795 1371226
14:10:06 2342912 52994 185772 2406 1397 435000 311843 1353500
14:22:30          unix restarts
Average 2342912 43246 70195 1904 925 233329 38919 1954394
$

```

The sar(1) output data columns have the following interpretation:

Column header	Interpretation
physmem	Physical pages of memory on system
kernel	Pages in use by the kernel
user	Pages in use by user programs
fsctl	Pages in use by file system to control buffers
fsdelwr	Pages in use by file system for delayed-write buffers
fsdata	Pages in use by file system for read-only data buffers
freedat	Pages of free memory that may be reclaimable
empty	Pages of free memory that are empty

Reporting Unused Memory Pages and Disk Blocks (sar -r)

The `sar(1)` command (with option `-r`) will report unused memory pages and disk blocks for the host system.

```
$ sar -r
IRIX64 flurry 6.5-ALPHA-1274427935 02241507 IP27 03/02/98
08:50:07 freemem freeswp  vswap
09:00:06 2270596 27611520 3083933
09:10:06 2262247 27611520 3085163
09:20:06 2294598 27611520 3085475
09:30:06 2218342 27611520 3081868
09:40:06 2167508 27611520 2890365
09:50:06 2074090 27611520 2879629
10:00:06 2071240 27611520 2905279
10:10:07 2059354 27611520 1313505
10:20:06 1878832 27611520 1336262
10:30:06 1858292 27611520 1335474
10:40:06 1832440 27611520 1296596
10:50:06 1805964 27611520 1301303
Average 2097120 27611520 2299571
$
```

The `sar(1)` output data columns have the following interpretation:

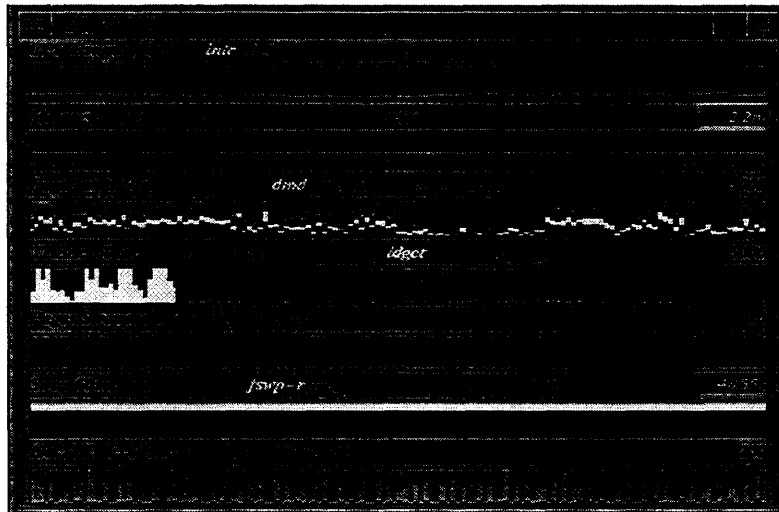
Column header	Interpretation
freemem	Average pages available to user processes
freeswap	Disk blocks available for process swapping
vswap	Virtual pages available to user processes

Reporting Memory Activity (`gr_osview(1)`)

The `gr_osview(1)` command will produce a graphical display of memory management activity including memory usage, page faults, TLB activity, and page swapping. An example of a `gr_osview(1)` display is shown below for a 128-CPU Origin2000 system with the user's `.grosview` file set to (see `gr_osview(1)` man page for details):

```
cpu(sum) strip creepscale
rmem strip creepscale interval(2)
fault strip creepscale colors(1,2,3,4,5,6,72,92)
tlb strip creepscale
pswap strip creepscale
swp strip creepscale
nettcp strip creepscale
```

System: Origin2000 system (128 CPUs)



Module 10: UNIX Filesystem Overview

UNIX Filesystem Overview

Unit covers:

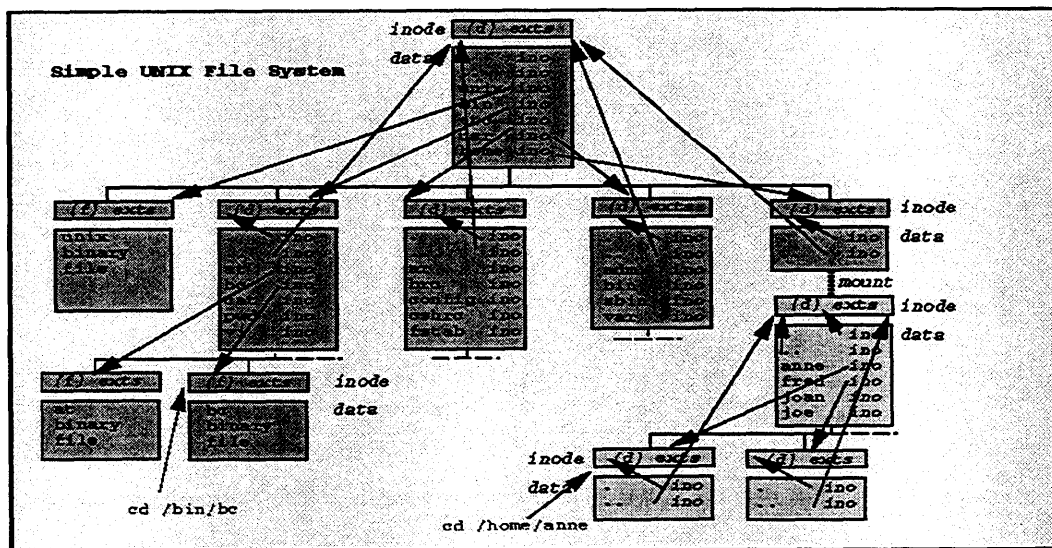
- Generic layout of a UNIX filesystem
- Layout of a UNIX System V filesystem
- Layout of an IRIX EFS filesystem (TBD)
- Layout of an IRIX XFS filesystem (TBD)

10-1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Sample UNIX File System



10-2

22jul1998

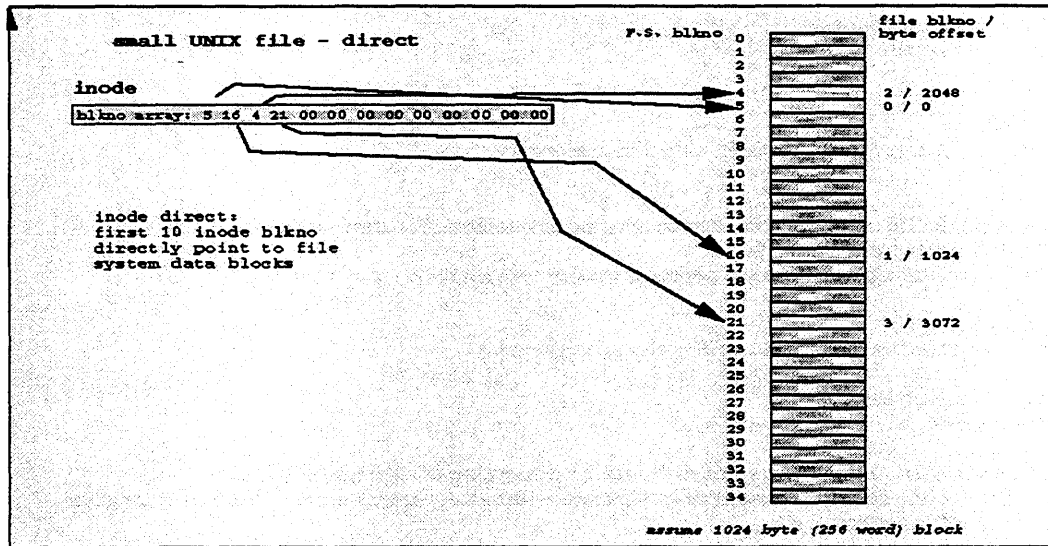
TR-IKI rev 0.7b SGI Proprietary

Generic UNIX FileSystem

- Hierarchy of regular and directory files
- File composed of inode and data
- Inode
 - File type
 - Access permissions
 - Pointers to where file's data "lives" on disk, called here extents (*exts*)
- Directory files
 - Inode type directory (*d*)
 - File data (contents) lists file names and corresponding inode numbers (locations on disk)
 - File name "." (dot) references inode of "self"
 - File name ".." (dot-dot) references inode of parent directory (except root - see below)
- Data files
 - Inode type regular file (*f*)
 - Data appears as sequential or random list of file characters (bytes)
- Root directory "/"
 - Topmost directory in filesystem
 - Parent directory is itself
- Mount point
 - Several filesystems may be "joined" together to form (the perception of) a single filesystem
 - Root directory of one filesystem is mounted to (associated with) an otherwise empty directory in another (previously mounted) filesystem

UNIX System V filesystem

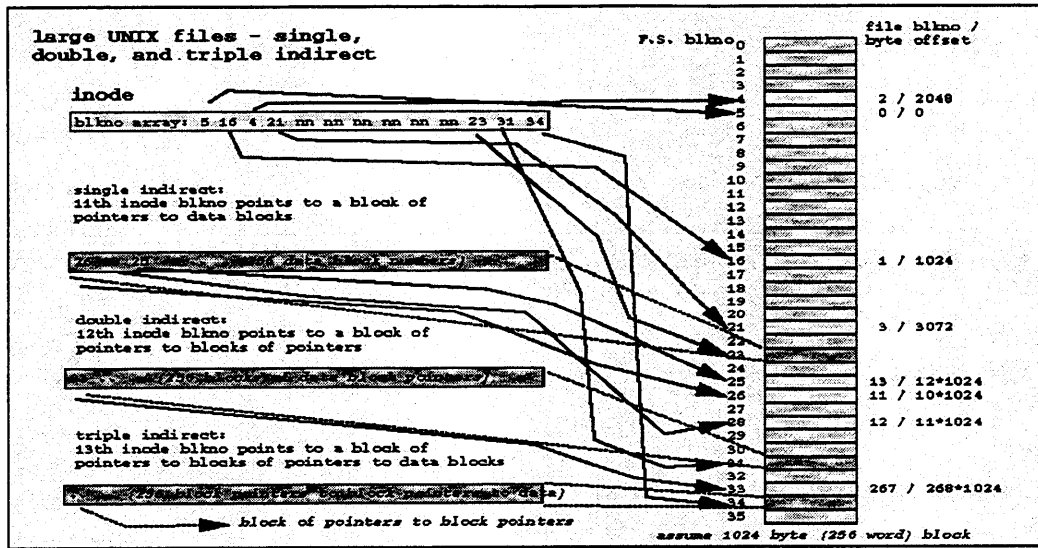
Small UNIX file sample



Small UNIX file

- Filesystem made up of blocks 0 through n (filesystem relative block number)
In sample, each block is 1024 bytes (256 four byte words)
- Inode contains block number (blkno) array
 - First 10 array items point directly to a file's data blocks
 - Array indices 11-13 are used for indirect descriptors (see next subject)
- Sample file
 - Composed of four filesystem blocks, in order 5, 16, 4, 21
 - Each f.s. block holds 1024 bytes user data
User sees data as a stream of (up to) 4096 characters; file size in inode
 - For example, f.s. blkno 21 is user block 4 (or character positions 3072-4095)

Large UNIX file sample



Large UNIX file

- Single indirect
 - When the file grows beyond what fits in ten direct block descriptors the file continues to grow using single (level) indirect
 - Inode array index 11's block is a block (full of) data block descriptors each pointing to data blocks
 - Sample
 - Inode element 11 points to f.s. block 23
 - F.S. block 23 holds 256 block descriptors each pointing to a 1024 byte block
 - File capacity is $10 + 256 = 266$ blocks
- Double indirect
 - When the file grows beyond what fits in the single in direct block capacity the file continues to grow using double indirect
 - Inode array index 12's block is a block (full of) data block descriptors each pointing to another block of block descriptors each pointing to data blocks
 - Sample
 - Inode element 12 points to f.s. block 32
 - F.S. block 32 holds 256 block descriptors, each pointing to a 256 word block.
 - Each of those 256 (possible) descriptors points to blocks pointing to 256 data blocks
 - File capacity is $10 + 256 + 256 * 256 = 65802$ blocks
- Triple indirect
 - In the extreme case when the file grows beyond what fits in the double indirect block capacity the file continues to grow using triple indirect
 - Inode array index 13's block is a block (full of) data block descriptors each pointing to another block of block descriptors each pointing to 1024 byte data blocks
 - Sample
 - Inode element 13 points to f.s. block 34
 - F.S. block 34 holds 256 block descriptors.
 - Each of those points to 256 blocks of block descriptors.
 - Each of those $256 * 256$ blocks points to 256 blocks of block descriptors

- Each of the $256*256*256$ blocks points to 1024 byte data blocks.
- File capacity is $10+ 256+256*256+256*256*256 = 16,843,018$ blocks

10-8.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Module 11: XFS Filesystem - Structure

The Extent Filesystem (EFS)

Limitations:

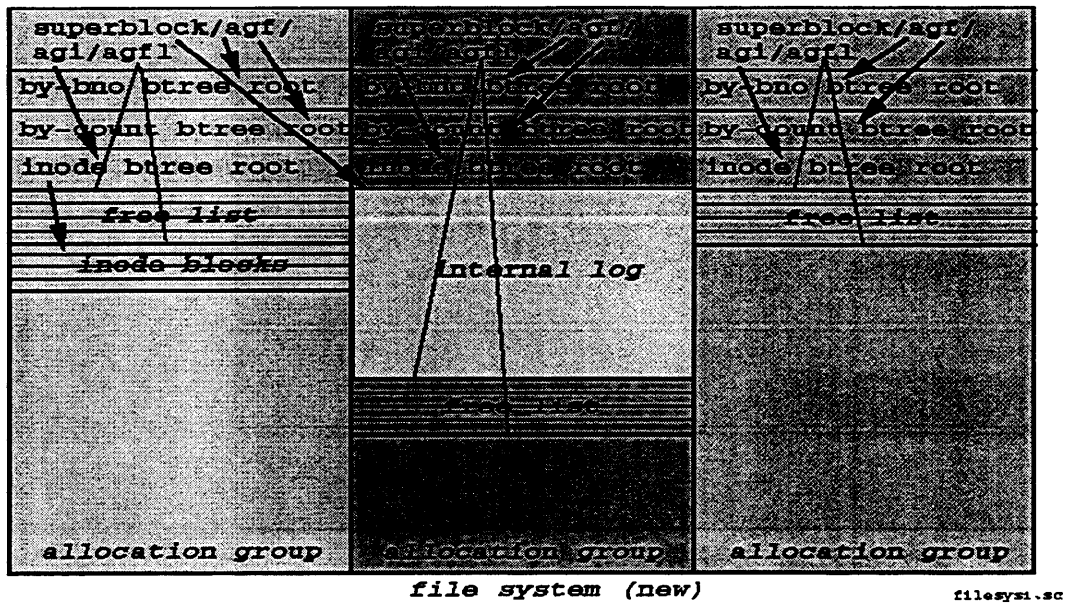
- Filesystem max: 8GB
- File max: 2GB
- Number of files fixed at mkfs time
- Less than full hardware bandwidth
- Slow crash recovery (minutes)
- No support of sparsely-allocated files
- Slow performance for large files
 - linear bitmap structures for tracking free space made finding contiguous space slow
 - linear searching of large directories

xFS: the extension of EFS

The main ideas:

- "x" for to-be-determined (but the name stuck)
- large filesystems
- large files
- large number of inodes
- large directories
- large I/O
- parallel access to inodes
- binary tree algorithms for searching large lists
- asynchronous metadata transaction logging for quick recover
- delayed allocation to improve data contiguity
- ACL's --Access Control Lists (see `chacl(1)`, `acl(4)`, `acl_get_file(3c)`, `acl_set_file(3c)`)

A New XFS Filesystem:



- 3 "allocation groups"
 - AG's are used to keep the size of freespace and inode management data structures manageable

- These structures use AG-relative block and inode pointers
- Directories are round-robined thru the AG's
- Files cluster around their directory, but are not limited to a single AG
- Free space and inode structures in memory are locked per-AG, so parallelism can be achieved filesystem wide
- built with a journaling log within the data fork (i.e. not an XLV logical volume with "data" "realtime" and "log" forks)
- mkfs -b size=blocksize \
 - d name=special-file,agcount=3 \ /* "data" section: device to build on, count of allocation groups*/
 - i size=256,maxpct=25 \ /* "inode" section: inode size, percent of fs for inodes */
 - l internal=1,size=512b /* "log" section: log is internal with inodes/data, size in blocks */
- example on a regular file:

```
mkfs -b size=4096 \
-d name=/tmp/cpw.fs,file=1,size=145133568,agcount=5 \
-i size=256,maxpct=25,align=1 \
-l internal=1,size=512b

meta-data= /tmp/cpw.fs   isize=256   agcount=5, agsize=7087 blks
data      =              bsize=4096  blocks=35433, imaxpct=25
          =              sunit=0         swidth=0 blks
log       = internal log bsize=4096  blocks=512
realtime  = none         extsz=65536 blocks=0, rtextents=0
```

Allocation Group:

— always 1 sector

superblock (512 bytes)	> sb [n] > print
agf: allocation group free blocks (512 bytes)	> agf [n] > print
agi: allocation group inode btree (512 bytes)	> agi [n] > print
agfl: allocation group internal free list (512 bytes)	> agfl [n] > print
free space btree - by block number	> sbblock n > type inobt > print
free space btree - by block count	> sbblock n > type inobt > print
inode btree	> sbblock n > type inobt > print

allocation group

sb2.sc

- superblock: identical one in each allocation group; describes basic characteristics of the filesystem and location of some key components

- agf: allocation group free space; points to the structures for locating free space on the filesystem
- agi: allocation group inodes; points to the structures for locating inodes in the allocation group
- agfl: used by XFS internally

Superblock:

```
sb_magicnum    magic number "XFSE"  
sb_blocksize   block size (bytes)  
sb_dblocks     number of data blocks  
sb_logstart    starting block of internal log  
sb_rootino     root inode number  
sb_agblocks    size of allocation group  
sb_agcount     number of allocation groups  
sb_logblocks   number of log blocks  
sb_inodesize   inode size (bytes)  
sb_inopblock   inodes per block  
sb_icount      allocated inodes  
sb_ifree       free inodes  
sb_fdblocks    free data blocks  
...
```

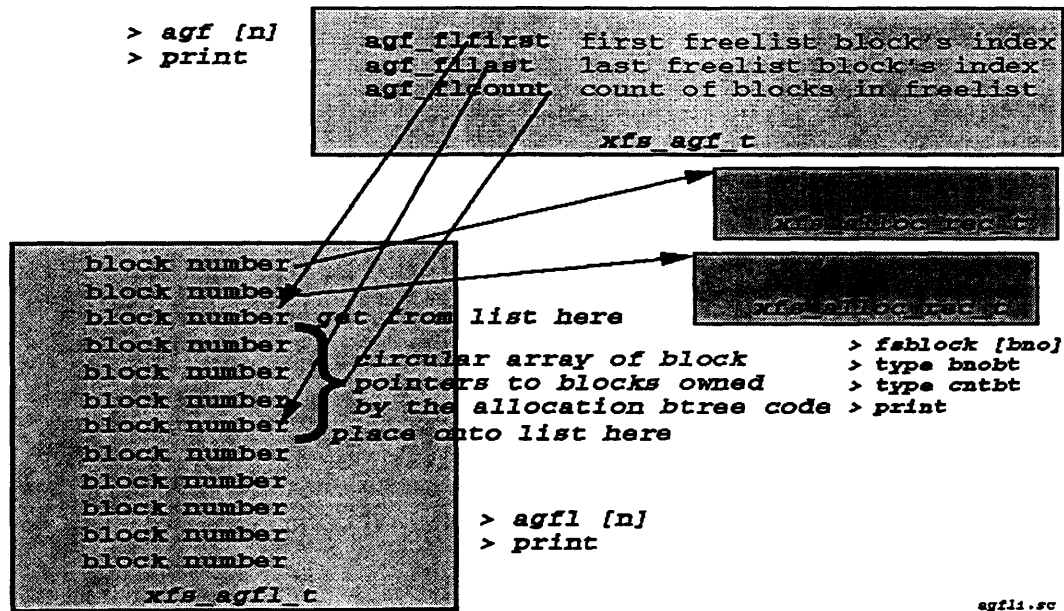
```
> sb [n]                xfs_sb_t                sbi.sc  
> print
```

- HREF="xfsdb.htm" TARGET="_blank">xfs_db "man page"
- xfs_db example:

```
xfs_db: sb  
xfs_db: print  
magicnum = 0x58465342  
blocksize = 4096  
dblocks = 35433
```

```
...  
logstart = 16388  
rootino = 128  
...  
agblocks = 7087  
agcount = 5  
...  
logblocks = 512  
...  
inodesize = 256  
inopblock = 16  
...  
imax_pct = 25  
icount = 64  
ifree = 61  
fdblocks = 34897  
...
```

AGFL - Allocation Group Free List:



• The Allocation Group Free List is only used internally by XFS to control agf btree blocks

Located in the 4th 512 byte block of each allocation group the agfl freelist for internal btree space allocation is maintained for each allocation group. This acts as a reserved pool of space separate from the general filesystem freespace (not used for user data)

- xfs_db "man page"
- xfs_db example:

```
xfs_db: agfl 0
xfs_db: print
bno[0-127] = 0:4 1:5 2:6 3:7 /* blocks 4, 5, 6 and 7 are occupied */
```

AGI: Inode Btree Control:

```
agi_magicnum    magic number "XAGI"
agi_seqno       sequence number, starting from 0
agi_length      blocks in the allocation group
agi_count       number of allocated inodes
agi_root        block number of root of inode btree
agi_level       levels in inode btree
agi_freecount   number of free inodes
agi_newino      new inode - just allocated
agi_dirino      last directory inode chunk
agi_unlinked[64] hash table of unlinked inodes (but
still being referenced)
```

xfs_agi_t

```
> agi [n]
> print
```

agi.sc

- xfs_db "man page"
- xfs_db example: [empty filesystem]

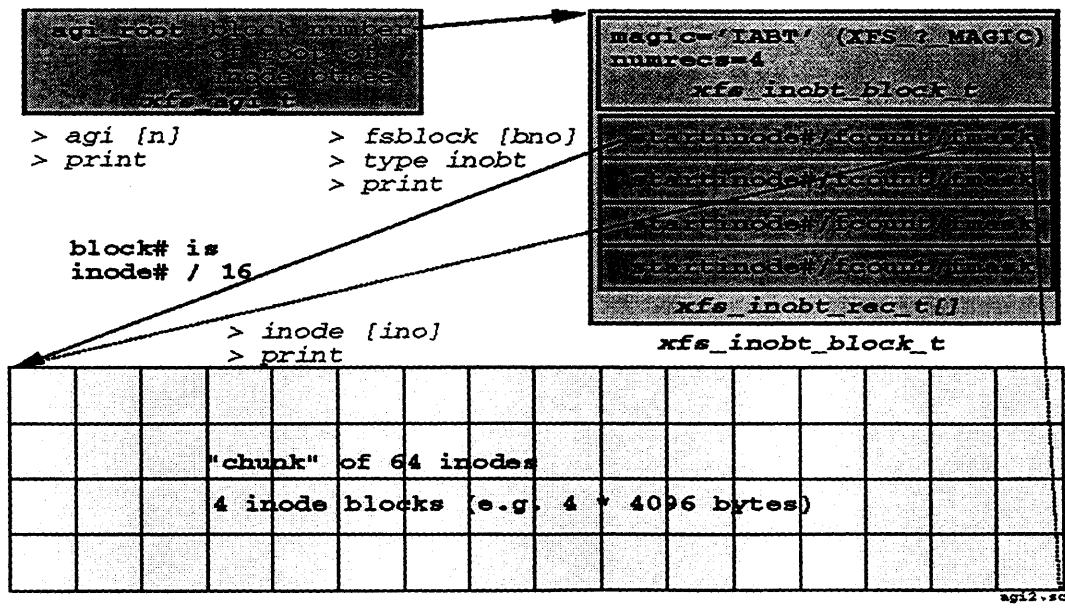
```
xfs_db: agi 0
xfs_db: print
magicnum = 0x58414749
versionnum = 1
seqno = 0
length = 7087
count = 64 /* allocated inodes in this a.g. */
```

```

root = 3          /* block number of root of inode btree */
level = 1
freecount = 61   /* free inodes in the a.g. */
newino = 128
dirino = null
unlinked[0-63] =

```

AGI and Inode Btree:



- XFS dynamically allocates inodes as needed
- XFS has replaced free space bit maps with a binary trees

- each entry in the inode btree block represents a chunk of 64 inodes (the above binary tree has only one level)
- xfs_db "man page"
- xfs_db example:

```

xfs_db: agi 0
xfs_db: print
...
count = 64
root = 3
level = 1
freecount = 61
...
xfs_db: fsblock 3
xfs_db: type inobt
xfs_db: print
magic = 0x49414254
level = 0
numrecs = 1
leftsib = null
rightsib = null
recs[1] = [startino, freecount, free] 1:[128, 61, 0xffffffffffff] /* inodes 131-191 are free */
/* note how the bit map represents low numbered inodes on the right */
/* from superblock: rootino = 128 (root) rbmino = 129 rsumino = 130 (for realtime extents) */

xfs_db: inode 128
xfs_db: print
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 1 (local)
core.nlinkvl = 2
core.uid = 0
core.gid = 0
core.atime.sec = Fri Feb 27 15:56:39 1998
core.atime.nsec = 931142000
core.mtime.sec = Fri Feb 27 15:56:39 1998
core.mtime.nsec = 931142000
core.ctime.sec = Fri Feb 27 15:56:39 1998
core.ctime.nsec = 931142000
core.size = 9
core.nblocks = 0
core.extsize = 0
core.nextents = 0

```

```

core.naextents = 0
core.forkoff = 0
core.aformat = 2 (extents) /* this inode will have extents -- currently are none */
...

```

On-disk Inode:



```

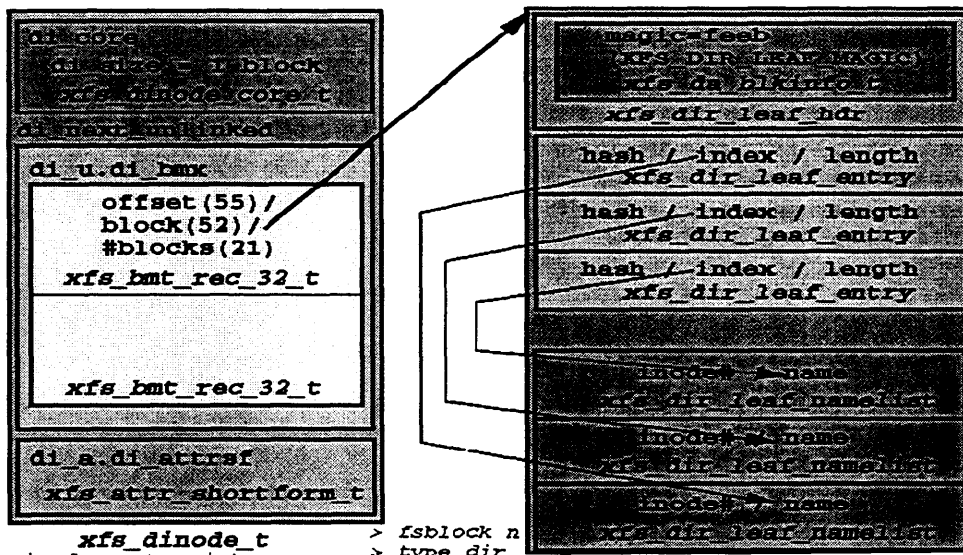
u.sfdir.list[3].inumber = 8388838
u.sfdir.list[3].namelen = 12
u.sfdir.list[3].name = "libmalloc.so"
u.sfdir.list[4].inumber = 8609299
u.sfdir.list[4].namelen = 3
u.sfdir.list[4].name = "cpp"
    
```

11-12.b

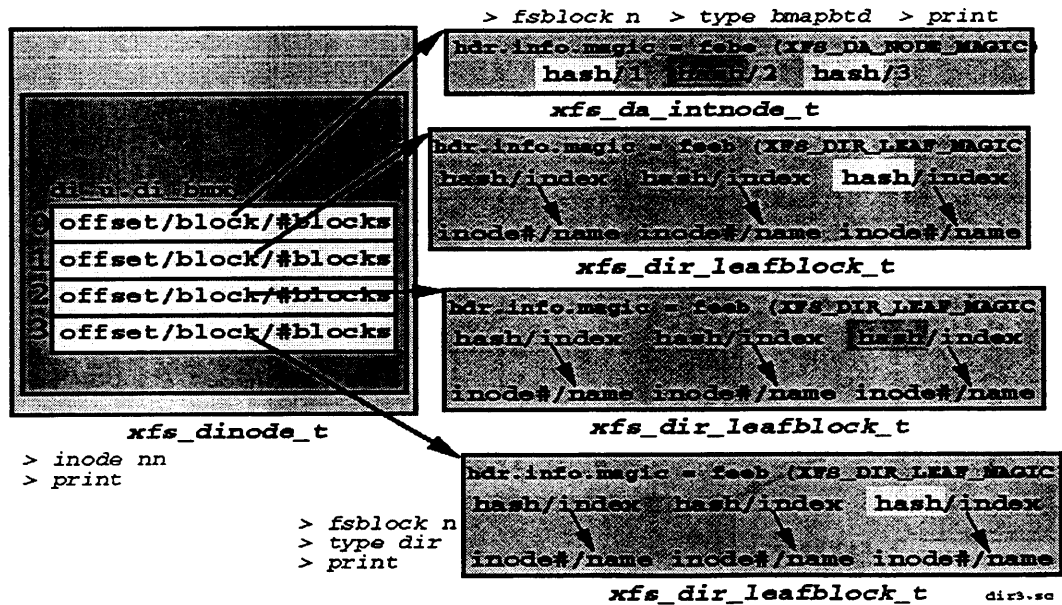
22jul1998

TR-IKI rev 0.7b SGI Proprietary

1-block Directory:



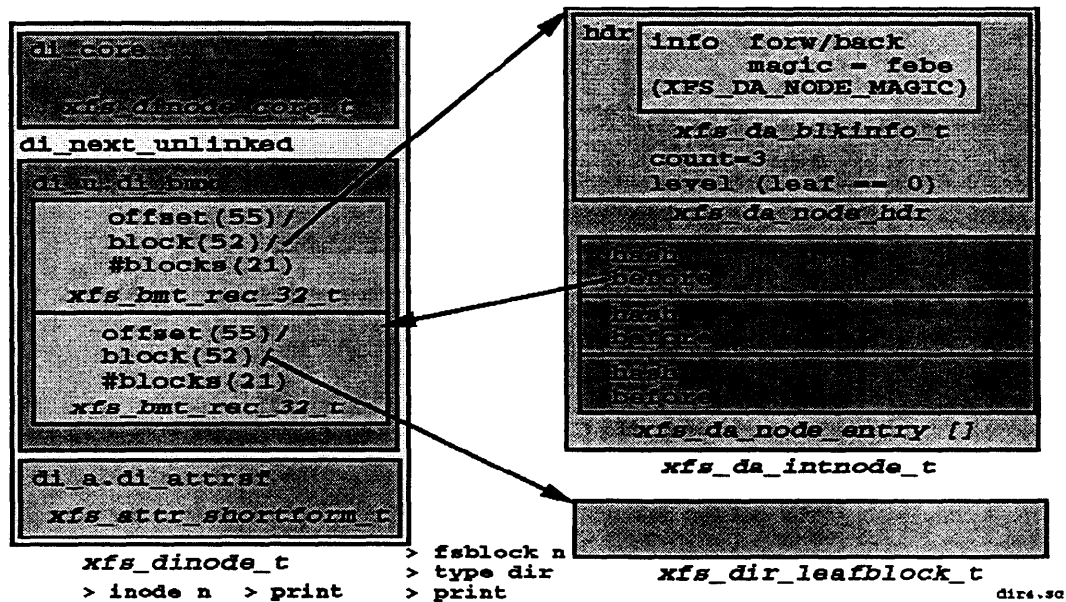
Btree Directory:



- pictured above is a 1-level binary tree structure for a directory
- the first extent is not directory data, but a table that tells what extent contains a given key

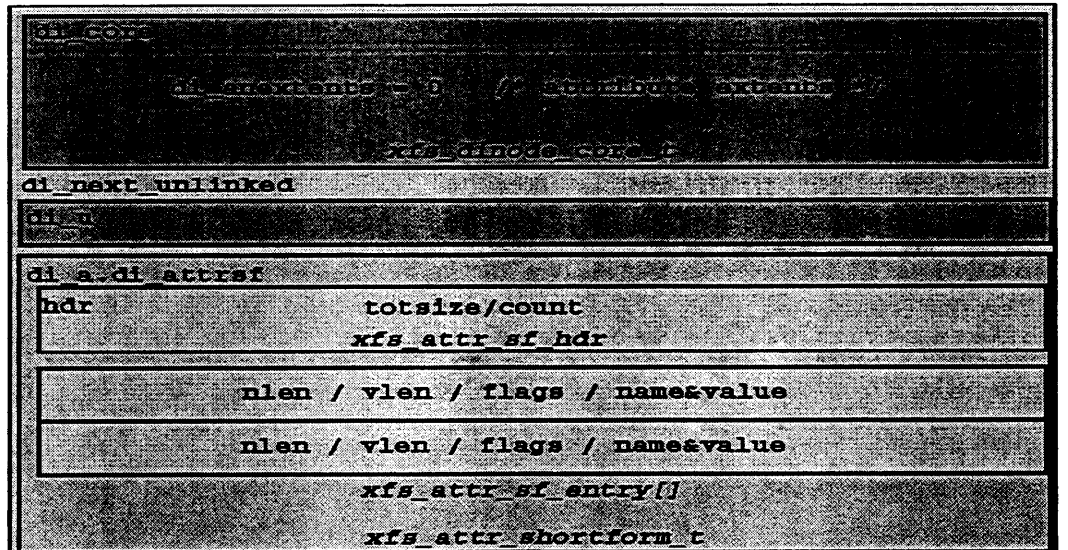
- any link with hash value up to the first entry would be found in extent[1] of the directory
- any link with hash value up to the second entry would be found in extent[2] of the directory
- each "leaf" of the directory contains a list of hash values in association with an index to the full link name within the block
- there may be duplicate hash keys

Btree Directory - Index Block:



- The above diagram provides a little more detail of the "xfs_da_intnode_t" block
- The "before" field is the index into the inode extents

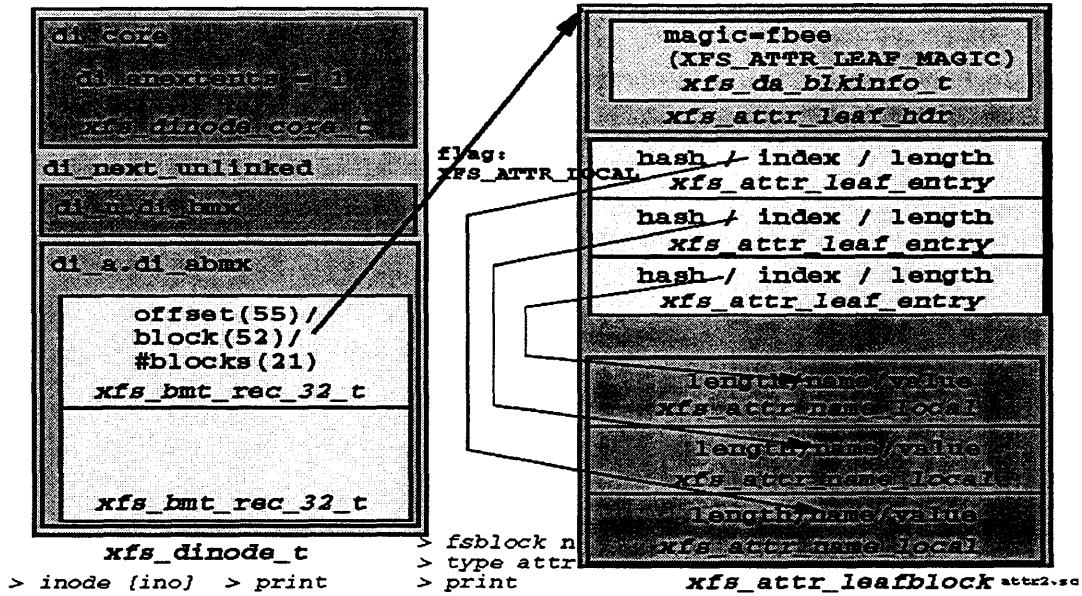
Attribute Fork Inside Inode:



> inode [ino] > print xfs_dinode_t attri.sc

- A small set of "attributes" may be stored entirely within an inode
- A target set will be stored separately, as shown below

Attributes Block:



- Access Control Lists (ACLs) are stored as attributes
- DMF bitmapped file id's are stored as attributes
 - it is highly recommended that filesystems to be DMF-managed be made with 512-byte inodes so that the DMF bfid's can be

stored with in the inodes

- The attributes of a file may be anything that its owner wishes to store with it
- example:


```

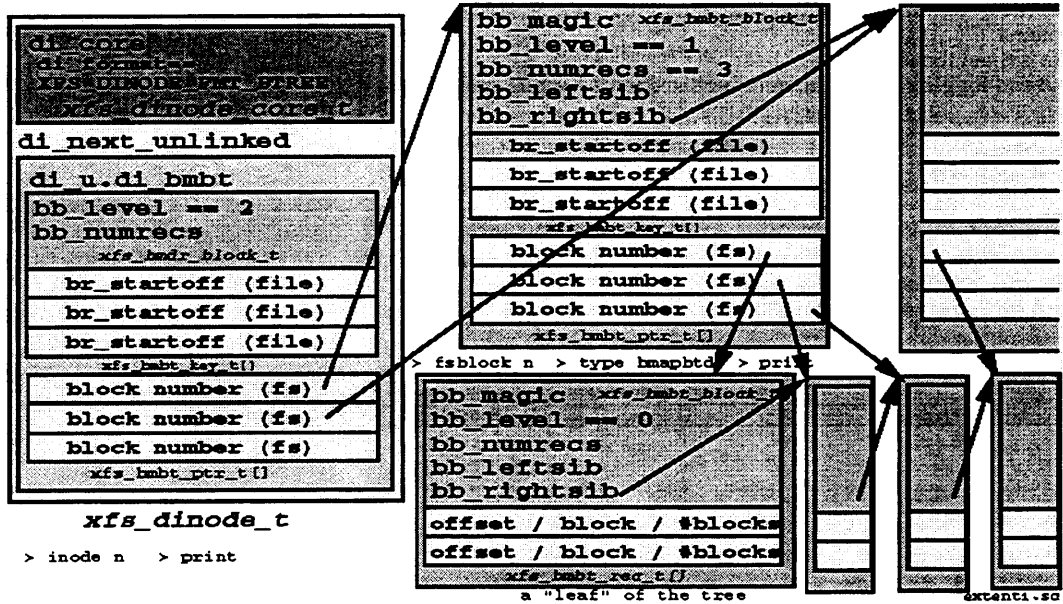
$ attr -s character_set -V kanji filex
Attribute "character_set" set to a 5 byte value for filex:
kanji

$ attr -s revision -V 5.1 filex
Attribute "revision" set to a 3 byte value for filex:
5.1

$ attr -l filex
Attribute "character_set" has a 5 byte value for filex
Attribute "revision" has a 3 byte value for filex

$ attr -g character_set filex
Attribute "character_set" had a 5 byte value for filex:
kanji
      
```

Data Fork - Binary Tree



- above is a file with a 2-level binary tree of extents
 - the inode points to blocks of indirect pointers

- the indirect blocks point to the leaf blocks containing the actual extent descriptions
- reference: reading an inode

```

xfs_iget()
xfs_iread()
xfs_ifformat
xfs_ifformat_btree
gets only the root of the btree into memory (attached to if_broot)
    
```

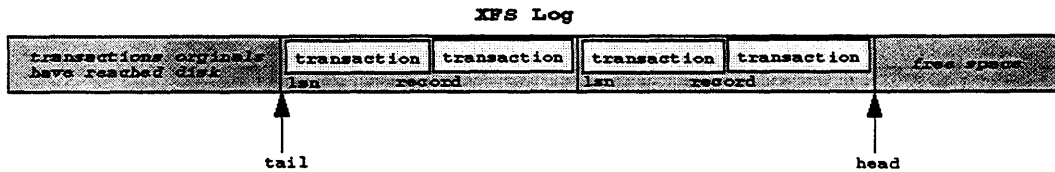
- reference: reading the entire extent list

```

xfs_iread_extents()
xfs_bmap_read_extents
reads all the extents (and attaches them to if_extents)
    
```

Journaling Log

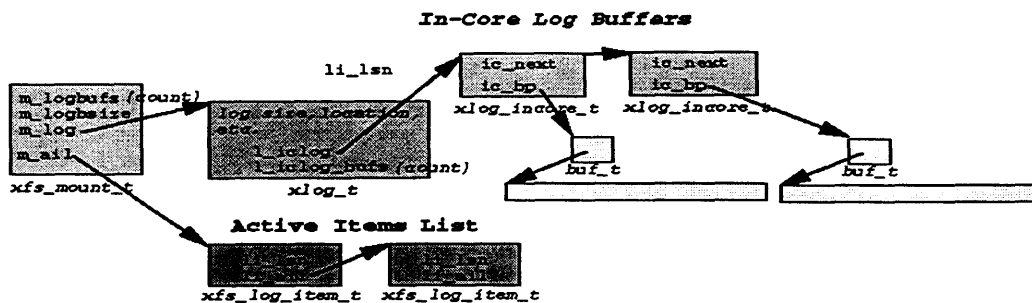
- Examining all the filesystem metadata to reconstruct it after a crash would take too long
 - large filesystems
 - inodes not in a fixed location
- XFS does write-ahead logging of all structural updates to filesystem metadata
 - inodes
 - directory blocks
 - free extent tree blocks
 - inode allocation tree blocks
 - file extent map blocks
 - AG header blocks
 - the superblock
- log entries must be written to disk before the metadata itself reaches disk
- the log is circular, with a tail chasing a head
- each record has a Log Sequence Number (lsn)



11-19

22jul1998

TR-IKI rev 0.7b SGI Proprietary

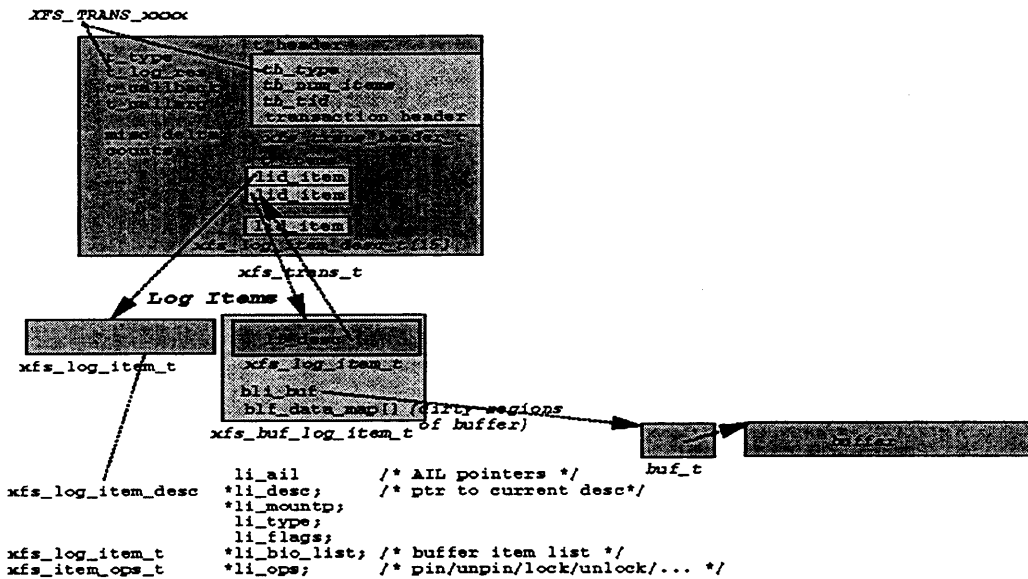


- The life cycle of a transaction:
 1. allocate a transaction structure in memory and assign it it a unique id (tid)
 - call xfs_trans_alloc()

11-19.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary



2. reserve disk space for the transaction
3. lock the metadata that is being logged; it must stay locked until the transaction is "committed"
4. modify the metadata resource and remember what was modified
 - call `xfs_trans_log_buf(xfs_trans_t *tp, buf_t *bp, uint first, uint last)` to remember specified bytes of a buffer
5. commit the transaction:
 - call `xfs_trans_commit(xfs_trans_t *tp, uint flags, xfs_lsn_t *commit_lsn_p)`
 - record the transaction (and all its modified metadata) in the in-core log (log records collected

- in one of 2 or more circular buffers)
 - "pin" the resources in memory until the transaction reaches disk (see `buf_t` field `b_pincount`)
 - unlock the resources
 - when the buffer is full, it is written as large, sequential write
 - 6. write of the transaction to disk completes:
 - handler was specified by call to `xfs_trans_callback()`
 - unpin the resources
 - free the transaction structure
 - modified resources associated with the transaction are placed in an Active Items List (AIL)
 - (there is one AIL per filesystem)
 - a metadata item stays "active" until it reaches disk
 - 7. modified resource (metadata) is flushed to disk (because of reuse, log buffer becomes nearly full, or some cleaning daemon pushes it out)
 - remove the item from the AIL (the log images are no longer needed)
- a log entry consists of a header describing the metadata image that follows, and a copy of that new image (see structure `xfs_buf_log_item`)
 - recovery consists of replaying the log
 - `xfs_repair(1M)` exists for correcting the results of errors that corrupt random blocks in the filesystem
 - `xfs_logprint(1M)` exists for displaying the contents of a log
 - example: change an inode with `chmod(2)`

```

chmod
set up a vattr structure with user's argument as mode and AT_MODE as mask

namesetattr(uap->fname, FOLLOW, &vattr, 0)

vp=lookupname(fnamep, UIO_USERSPACE, followlink, NULLVPP, &vp, NULL)

VOP_SETTATTR(vp, vop, flags, get_current_cred(), error)
xfs_setattr(bhv_desc_t *bdp, vattr_t *vap, int flags, cred_t *credp)

```

```

tp = xfs_trans_alloc(mp, XFS_TRANS_SETATTR_NOT_SIZE)
xfs_trans_reserve(tp, 0, XFS_ICHANGE_LOG_RES(mp), 0, 0, 0)
xfs_ilock(ip, lock_flags)
xfs_trans_ijoin(tp, ip, lock_flags)
if (ip->i_item == NULL)
    xfs_inode_item_init(ip, ip->i_mount)
    /* attach an item to the inode, if none already */
xfs_trans_add_item(tp, (xfs_log_item_t*)(iip))
    /* link inode's item to tp->t_items */
make user's requested changes to the inode
xfs_trans_commit(tp, commit_flags, NULL)
xfs_trans_count_vecs(tp)
    /* count items in the transaction
    IOP_SIZE(ip) calls xfs_inode_item_size() to count relevant inode
    pieces
xfs_trans_fill_vecs(tp, log_vector)
    /* fill in an array of iovecs for the transaction
    header and each item */
for the inode item: (and any other that exists)
IOP_FORMAT(lidp->lid_item, vecp) /* calls a format function */
xfs_inode_item_format(lidp->lid_item, vecp)
    /* sets i_addr and i_len to point to parts of inode to log */
IOP_PIN(lidp->lid_item) /* pins the inode in memory */
xfs_inode_item_pin(lidp->lid_item)
ip->i_pincount++
make the transaction header the first iovec
xfs_log_write(mp, log_vector, nvec, tp->t_ticket, &(tp->t_lsn))
    /* pass it the array of iovecs (i.e. address/length) */
xlog_write(mp, reg, nentries, tic, start_lsn, 0)
    /* pass it the array; write to in-core log */

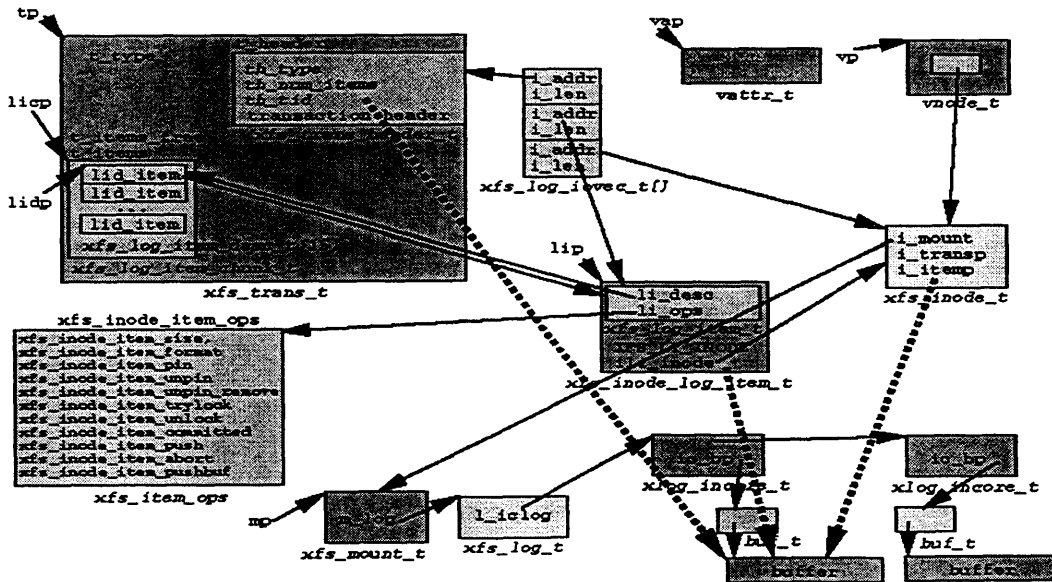
```

```

each item:
bcopy to m_log->log->buffers
xfs_trans_unlock_items(tp) /* free the "chunks" of each item */
xfs_iunlock(ip, lock_flags) /* unlock the inode */

```

• structures for the above example



● code references for write to log:

○ xfs_log_write->xlog_write->xlog_state_release_iclog->xlog_sync->bwrite

```

/*
 * Transaction types. Used to distinguish types of buffers.
 */
XFS_TRANS_SETATTR_NOT_SIZE 1
XFS_TRANS_SETATTR_SIZE 2
XFS_TRANS_INACTIVE 3
XFS_TRANS_CREATE 4
XFS_TRANS_CREATE_TRUNC 5
XFS_TRANS_TRUNCATE_FILE 6
XFS_TRANS_REMOVE 7
XFS_TRANS_LINK 8
XFS_TRANS_RENAME 9
XFS_TRANS_MKDIR 10
XFS_TRANS_RMDIR 11
XFS_TRANS_SYMLINK 12
XFS_TRANS_SET_DMATRS 13
XFS_TRANS_GROWFS 14
XFS_TRANS_STRAT_WRITE 15
XFS_TRANS_DIOSTRAT 16
XFS_TRANS_WRITE_SYNC 17
XFS_TRANS_WRITEID 18
XFS_TRANS_ADDAFORK 19
XFS_TRANS_ATTRINVAL 20
XFS_TRANS_ATRUNCATE 21
XFS_TRANS_ATTR_SET 22
XFS_TRANS_ATTR_RM 23
XFS_TRANS_ATTR_FLAG 24
XFS_TRANS_CLEAR_AGI_BUCKET 25
XFS_TRANS_QM_SBCHANGE 26

```

Sequence for replaying the log when the filesystem is mounted:

```

xfs_mountfs_int(vfsp, mp, dev, 1, 0)

xfs_log_mount(mp, logdev, XFS_FSB_TO_DADDR(mp, logstart),
             XFS_FSB_TO_BB(mp, sbp->sb_logblocks))

xlog_recover(log, XFS_MTOVFS(mp)->vfs_flag & VFS_RDONLY)
xlog_do_recover(log, head_blk, tail_blk)
xlog_do_log_recovery(log, head_blk, tail_blk)

xlog_do_recovery_pass(log, head_blk, tail_blk, XLOG_RECOVER_PASS1)
/* builds a hashed list of xlog_recover_t structures from the
   transactions in the log */
xlog_do_recovery_pass(log, head_blk, tail_blk, XLOG_RECOVER_PASS2)

xlog_bread(log, blk_no, 1, hbp) /* read the header block */
xlog_bread(log, blk_no+1, bblks, dbp) /* read the data block */
xlog_recover_process_data(log, rhash, rhead, dbp->b_dmaaddr, pass)

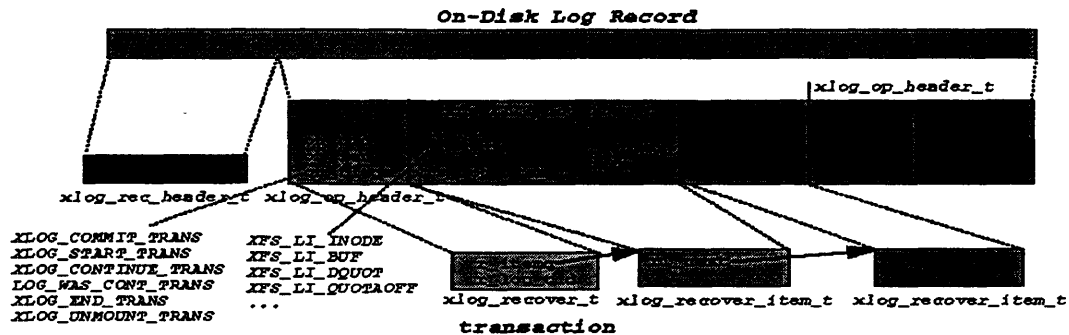
/* depending on transaction type; e.g. */
case XLOG_COMMIT_TRANS: {
xlog_recover_commit_trans(log, &rhash[hash], trans, pass)

xlog_recover_do_trans(log, trans, pass)

/* depending on item type; e.g. for an inode: */
if ((ITEM_TYPE(item) == XFS_LI_INODE)) {
xlog_recover_do_inode_trans(log, item, pass))

xfs_bwrite(mp, bp) /* write the metadata image to disk */

```



- Each log record has a header of one sector (xlog_rec_header_t)

I/O Performance

The main ideas behind XFS increase in I/O performance are:

- allocate large (contiguous) extents for files
 - writes to buffer cache reserve "virtual extents"; real blocks are not assigned until buffers are flushed to disk
 - short-lived files may never be allocated; their metadata updates are reduced
 - randomly-written files (with no holes) will be contiguous
 - a filesystem's block size may range from 512-bytes to 64KB; large blocks reduce fragmentation
- perform I/O in parallel
 - clustering - large minimum read buffers; combining writes of dirty buffers into large chunks
 - read ahead - multiple (2-3) read ahead buffers
 - write behind - balanced buffering of dirty blocks and asynchronous flushing to disk
 - request parallelism - multiple processes are allowed read/write the same file (inode lock) at the same time
 - IRIX supports asynchronous I/O by using multiple threads
 - direct I/O - avoiding copy into/out of buffer cache; and allows program control of I/O requests
 - buffer cache is kept coherent
- handle metadata efficiently
 - write-ahead transaction log makes updates fast
 - gather multiple updates into single I/O
 - write to the log asynchronously (sync. if exported via NFS)
 - modified data cannot be written until the log is on-disk
 - the log may be placed on a separate device (XLV "fork")
 - do searches and updates faster than linearly
 - allow parallelism
 - all resources (except the log) are independent across AG's or individual inodes
 - inodes and blocks can be allocated and freed in parallel

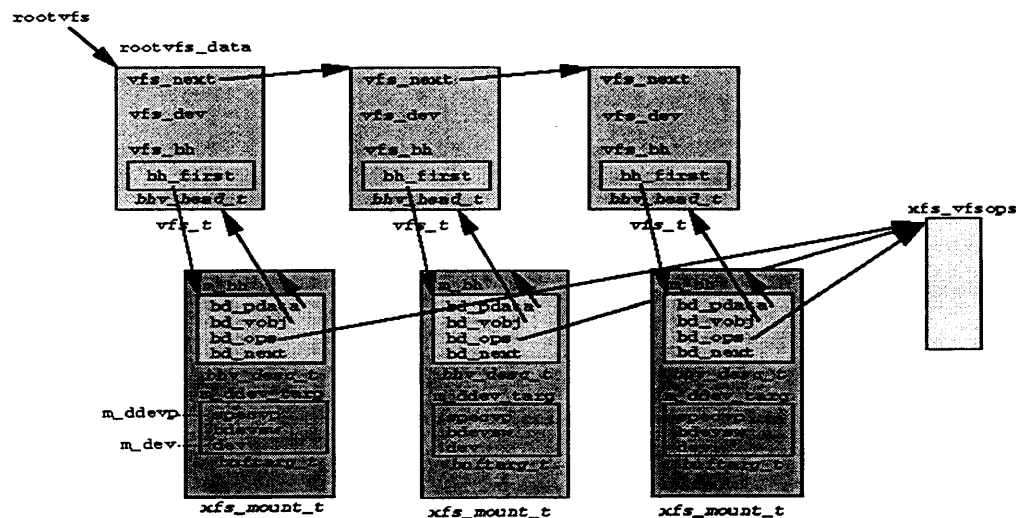
xfs_db printable block types:

```

xfs_db "man page"
sb      superblock
agf     ag freespace control
agi     ag inode control
agfl    ag freelist
bnobt   block of freespace btree sorted by block number
cntbt   block of freespace btree sorted by count
inobt   block of inode btree
dir     directory leaf block
inode   inode
bmapbtd block of an inode's extent btree for the data fork
bmapbta block of an inode's extent btree for the attribute fork
attr    attribute leaf block
dqblk   disk quota block
symlink symbolic link
data    hex dump of the block

```

Mounted Filesystems



- the inode points to the mount structure for the filesystem on which it resides
- the mount table points to a table of functions that do operations on the filesystem
- the mount table is dynamically allocated and "virtualized" - i.e. a filesystem-independent vfs_t structure points to a filesystem dependent structure (in the diagram, the xfs_mount_t)
- view the root vfs with icrash:

```

>> whatis rootvfs_data
0xc00000000140d800

>> print *(struct vfs *)0xc00000000140d800
struct vfs {
  vfs_next = 0xa8000000124289f00
  vfs_prevp = (nil)
  vfs_vnodecovered = (nil)
  vfs_dcount = 0
  vfs_flag = 6176
  vfs_dev = 1457
  vfs_nsubmounts = 24
  vfs_buyscnt = 0
  vfs_wait = sv_t {
    sv_queue = 0
  }
  vfs_bsize = 4096
  vfs_fstype = 1
  vfs_fsid = fsid_t {
    val = {
      [0] 773584821
      [1] 3197743642
    }
  }
  vfs_bcount = 0
  vfs_mac = (nil)
  vfs_acl = (nil)
  vfs_cap = (nil)
  vfs_inf = (nil)
  vfs_altfsid = 0xa80000000168354c
  vfs_bh = bhv_head_t {
    bh_first = 0xa800000001683300 /* this points to the bhv_desc in the mount structure */
  }
}

```

● view all devices in the mount table

```

>> dump rootvfs
>> walk -s vfs vfs_next c00000000140d800 | cat > vfss
>> sh
$ grep vfs_dev vfss
  vfs_dev = 61
  vfs_dev = 1457

```

11-23.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

  vfs_dev = 1677
  vfs_dev = 1580
  vfs_dev = 1540
  vfs_dev = 1510
  vfs_dev = 1486
  vfs_dev = 1546
  vfs_dev = 1516
  vfs_dev = 1574
  vfs_dev = 1492
  vfs_dev = 1131
  vfs_dev = 1608
  vfs_dev = 1557
  vfs_dev = 1649
  vfs_dev = 1790
  vfs_dev = 12582912 binary 110000 000000000000000000
  vfs_dev = 11272194
  vfs_dev = 11272193
  vfs_dev = 11272192
  vfs_dev = 13369344
  vfs_dev = 12845056
  vfs_dev = 13631488
  vfs_dev = 50331656
  vfs_dev = 50331652
  vfs_dev = 50331653
  vfs_dev = 50331654
  vfs_dev = 50331655

```

The low numbers are hwgraph handles.
The higher numbers (with bits above 17) are major/minors. (192 are XLV's)
\$ dtb 50331655
11000000 00000000000000000111
\$ btd 11000000
192

● view the root mount structure with icrash

```

>> print *(bhv_desc_t *)0xa800000001683300
struct bhv_desc {
  bd_pdata = 0xa800000001683300 /* this points to the beginning of the mount structure */
  bd_vobj = 0xc00000000140d800
  bd_ops = 0xc0000000013be008
  bd_next = (nil)
}

```

11-23.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

>> print *(xfs_mount_t *)0xa80000001683300
struct xfs_mount {
  m_bhv = bhv_desc_t {
    bd_pdata = 0xa800000001683300
    bd_vobj = 0xc00000000140d800
    bd_ops = 0xc0000000013be008
    bd_next = (nil)
  }
  m_tid = 0
  m_ail_lock = 2
  m_ail = xfs_ail_entry_t {
    ail_forw = 0xa80000009003e3540
    ail_back = 0xa80000003003fb540
  }
  m_ail_gen = 2457
  m_sb = xfs_sb_t {
    sb_magicnum = 1481003842
    sb_blocksize = 4096
    sb_dblocks = 979483
    sb_rblocks = 0
    sb_rextents = 0
    sb_uuid = uuid_t {
      __u_bits = {
        [0] 167
        [1] 118
        [2] 239
        [3] 181
        [4] 85
        [5] 151
        [6] 16
        [7] 32
        [8] 134
        [9] 165
        [10] 8
        [11] 0
        [12] 105
        [13] 2
        [14] 161
        [15] 250
      }
    }
    sb_logstart = 524292
    sb_rootino = 128
    sb_rbmino = 129

```

```

  sb_rsumino = 130
  sb_rextsize = 16
  sb_agblocks = 122436
  sb_agcount = 8
  sb_rmblocks = 0
  sb_logblocks = 1000
  sb_versionnum = 388
  sb_sectsize = 512
  sb_inodesize = 256
  sb_inopblock = 16
  sb_fname = ""
  sb_fpack = ""
  sb_blocklog = 12
  sb_sectlog = 9
  sb_inodelog = 8
  sb_inopblog = 4
  sb_agblklog = 17
  sb_rextslog = 0
  sb_inprogress = 0
  sb_imax_pct = 25
  sb_icount = 48128
  sb_ifree = 914
  sb_fdblocks = 495020
  sb_frextents = 0
  sb_uquotino = 0
  sb_pquotino = 0
  sb_qflags = 0
  sb_inoalignmt = 2
  sb_unit = 0
  sb_width = 0
}
m_sb_lock = 2
m_sb_bp = 0xa8000000015be700
m_fsname = 0xa800000000daff720 = /*
m_fsname_len = 2
m_dev = 1457 /* binary: 1011 0110001 */
m_logdev = 1457
m_rtdev = 0
m_bsize = 8
m_agfrotor = 0
m_agirotor = 2
m_ipinlock = 2
m_ihash = 0xa8000000016b4000
m_ihashmask = 1023

```

```

m_inodes = 0xa80000172384a600
m_ilock = mutex_t (
    m_bits = 0
    m_queue = (nil)
)
m_ireclaims = 0
m_readio_log = 16
m_readio_blocks = 16
m_writeio_log = 16
m_writeio_blocks = 16
m_log = 0xa800000015bea00
m_logbufs = -1
m_logbsize = -1
m_rsumlevels = 0
m_rsumsize = 0
m_rbmip = 0xa800000015ed000
m_rsumip = 0xa800000015ed200
m_rootip = 0xa800000015ece00
m_quotainfo = (nil)
m_ddevp = 0xa800000100499200
m_logdevp = 0xa800000100499200
m_rtdevp = (nil)
m_dircook_elog = 8
m_blkbit_log = 15
m_blkbb_log = 3
m_agno_log = 3
m_agino_log = 21
m_nreadaheads = 4
m_inode_cluster_size = 8192
m_blockmask = 4095
m_blockwsize = 1024
m_blockwmask = 1023
m_alloc_mxr = (
    [0] 510
    [1] 340
)
m_alloc_mnr = (
    [0] 255
    [1] 170
)
m_bmap_dmnr = (
    [0] 254
    [1] 254
)

```

11-23.e

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

m_bmap_dmnr = (
    [0] 127
    [1] 127
)
m_inobt_mxr = (
    [0] 255
    [1] 510
)
m_inobt_mnr = (
    [0] 127
    [1] 255
)
m_ag_maxlevels = 3
m_bm_maxlevels = (
    [0] 5
    [1] 3
)
m_in_maxlevels = 2
m_perag = 0xa80000000d9cc00
m_peraglock = mrlock_t (
    mr_lbits = 4
    mr_un = union {
        mr_st = struct {
            qcount = 0
            qflags = 0
        }
        qbits = 0
    }
    mr_queue = (nil)
)
m_growlock = sema_t (
    s_un = union {
        s_st = struct {
            count = 1
            flags = 0
        }
        s_lock = 65536
    }
    s_queue = (nil)
)
m_rbmrotor = 0
m_fixedfsid = (
    [0] -2035985001
    [1] -1485377611
)

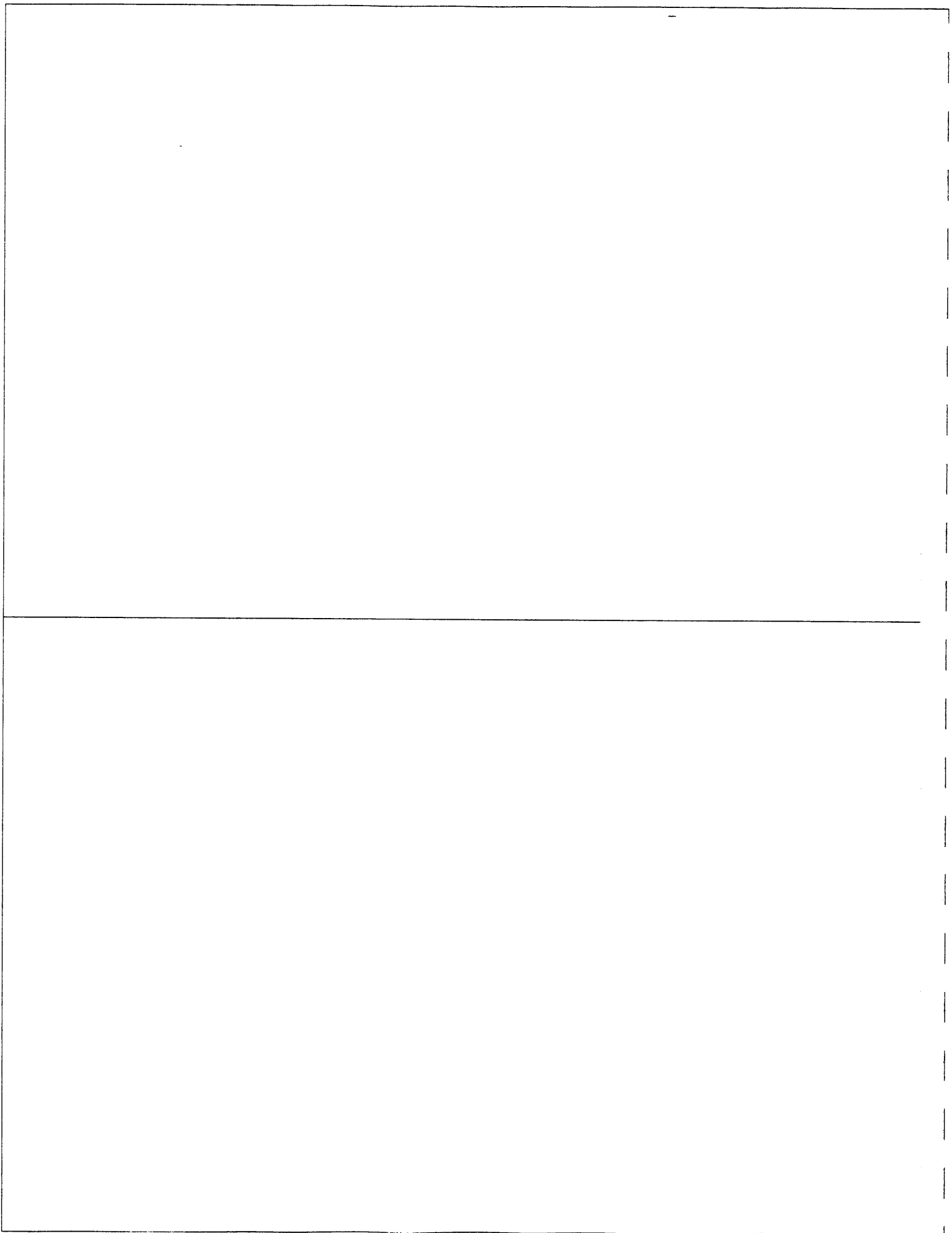
```

11-23.f

22jul1998

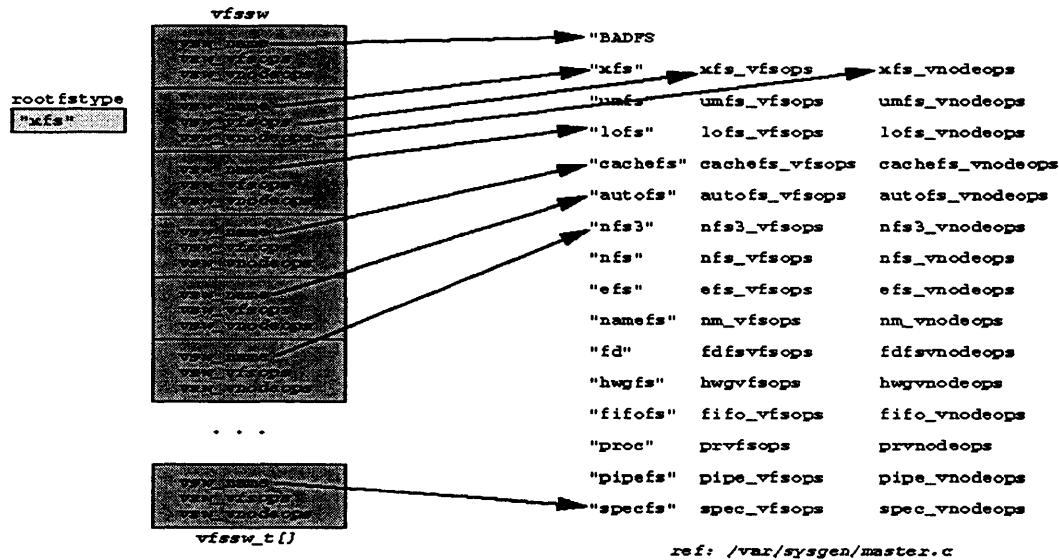
TR-IKI rev 0.7b SGI Proprietary

```
)
m_dmevmask = 0
m_flags = 0
m_attroffset = 120
m_da_node_ents = 510
m_ialloc_inos = 64
m_ialloc_blks = 4
m_litino = 156
m_inoalign = 2
m_qflags = 0
m_reservations = xfs_trans_reservations_t {
    tr_write = 74424
    tr_itruncate = 203704
    tr_rename = 86328
    tr_link = 43008
    tr_remove = 43320
    tr_symlink = 52280
    tr_create = 52280
    tr_mkdir = 52592
    tr_ifree = 14520
    tr_ichange = 1592
    tr_growdata = 27264
    tr_swrite = 384
    tr_addafork = 31416
    tr_writeid = 384
    tr_attrinval = 128000
    tr_attrset = 128312
    tr_attrrm = 128312
    tr_clearagi = 640
}
m_maxicount = 3917920
m_inoadd = 0
m_dalign = 0
m_swidth = 0
m_sinoalign = 0
}
```



Module 12: XFS File Management

File System Switch



- at boot time, the type of the root filesystem is taken from rootfstype
- for each filesystem type there are 2 central logic tables:
 - XXX_vfsops - functions that do operations on the filesystem (mount, unmount, sync, ...)
 - XXX_vnodeops - functions that do operations on files (open, close, read, write, ...)

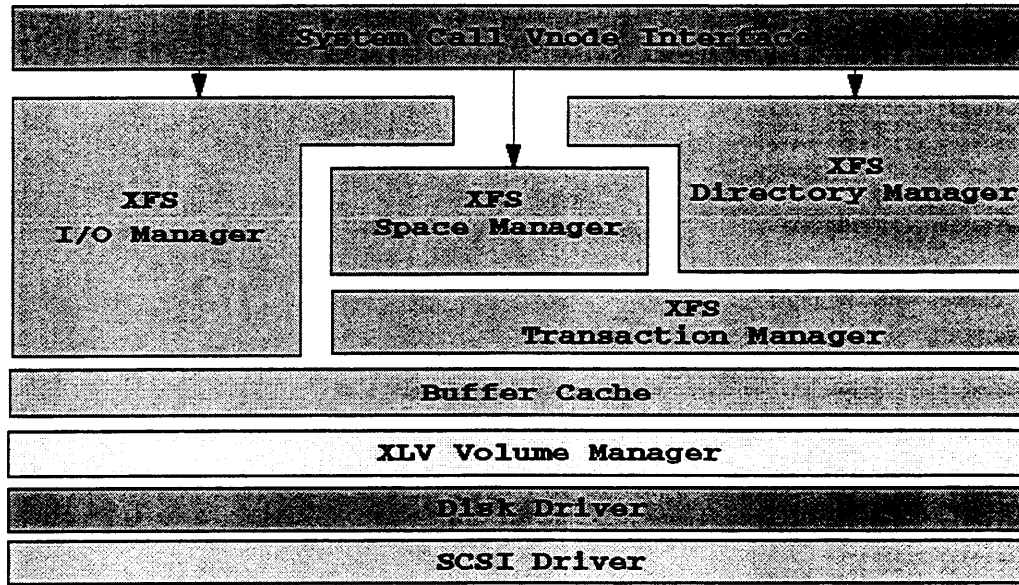
- the pointer to the XXX_vfsops is stored in the mount table whenever a filesystem is mounted
- the vfsops functions store the XXX_vnodeops pointer in the in-memory inode when it is allocated
- displaying the vfssw with icrash:

```
>> fstype
INDEX  NAME          INIT          VFSOPS          VNODEOPS        FLAG
-----
0      BADVFS        0             c000000001426ab0 c000000001426b80 0
1      xfs          c0000000002c5690 c00000000143a450 c00000000143a218 0
2      umfs        c0000000002e29fc c00000000143af10 c00000000143ad48 0
3      lofs        c0000000002eae0c c00000000143b090 c00000000143b108 0
4      cachefs     c000000000302020 c00000000143bd60 c00000000143be28 0
5      autofs     c00000000031c4dc c00000000143c1e0 c00000000143c018 0
6      nfs3       c00000000032d880 c00000000143c510 c00000000143c258 0
7      nfs        c0000000002ec55c c00000000143b2d0 c00000000143b348 0
8      efs        c000000000350994 c00000000143d190 c00000000143cfb0 0
9      namefs     c000000000353408 c00000000143d3d0 c00000000143d208 0
10     fd         c000000000354890 c00000000143d688 c00000000143d4c0 0
11     hwgfs     c000000000356610 c00000000143d8c8 c00000000143d700 0
12     fifofs    c000000000357388 c00000000143d940 c00000000143d9d8 0
13     proc      c00000000035bccc c00000000143dba0 c00000000143dc18 0
14     pipefs    c000000000365c74 c00000000143dde0 c00000000143de58 0
15     specfs    c0000000002dad74 c00000000143a870 c00000000143a6a8 0
-----
16 vfs structs found

>> findsym c00000000143a450
=====
0xc00000000143a450 --> xfs_vfsops
=====
1 symbol found

>> findsym c00000000143a218
=====
0xc00000000143a218 --> xfs_vnodeops
=====
1 symbol found
```

XFS Code Architecture



- XFS is called by POSIX-compliant system calls
 - uses buffer/page cache

- uses directory name lookup cache
- uses dynamic vnode cache
- Space Manager
 - manages filesystem free space
 - manages allocation of inodes
 - manages allocation of space within files
- I/O Manager
 - satisfies file I/O requests
- Directory Manager
 - implements "name space" (addressing of objects by name)
- Transaction Manager
 - updates all filesystem metadata (inodes, superblock, agf,...)
- Buffer Cache
 - frequently accessed blocks from the underlying volumes
 - integrated memory pages and all-filesystem cache
- XLV Volume Manager
 - provides disk concatenation, striping and mirroring (plexing)
- display defined logical volumes with xlv_mgr:

```

xlv_mgr> show -long all
VOL src_a (complete)          (node=flurry)
VE src_a.data.0.0             [active]
    start=0, end=8876254, (cat)grp_size=1
    /dev/dsk/dks6d1s7 (8876255 blks)
VE src_a.data.0.1             [active]
    start=8876255, end=17752509, (cat)grp_size=1
    /dev/dsk/dks6d2s7 (8876255 blks)

VOL tmp (complete)           (node=flurry)
VE tmp.data.0.0 [active]
    start=0, end=302076671, (stripe)grp_size=17, stripe_unit_size=256
    /dev/dsk/dks6d3s7 (17777424 blks)
    /dev/dsk/dks6d4s7 (17777424 blks)
    /dev/dsk/dks12d3s7 (17777424 blks)
    /dev/dsk/dks12d4s7 (17777424 blks)
    
```

```
/dev/dsk/dks14d3s7 (17777424 blks)
/dev/dsk/dks14d4s7 (17777424 blks)
/dev/dsk/dks20d3s7 (17779016 blks)
/dev/dsk/dks20d4s7 (17779016 blks)
/dev/dsk/dks22d3s7 (17777424 blks)
/dev/dsk/dks22d4s7 (17777424 blks)
/dev/dsk/dks30d4s7 (17769232 blks)
/dev/dsk/dks36d4s7 (17769232 blks)
/dev/dsk/dks38d4s7 (17769232 blks)
/dev/dsk/dks44d4s7 (17769232 blks)
/dev/dsk/dks46d4s7 (17769232 blks)
/dev/dsk/dks52d4s7 (17769232 blks)
/dev/dsk/dks54d4s7 (17769232 blks)
```

```
VOL ptmp (complete) (node=flurry)
VE ptmp.data.0.0 [active]
start=0, end=17769231, (cat)grp_size=1
/dev/dsk/dks12d5s7 (17769232 blks)
VE ptmp.data.0.1 [active]
....
```

● Show active logical volumes with xlv_mgr:

```
xlvmgr> show kernel
```

```
VOL tmp flags=0x1, [complete] (node=NULL)
DATA flags=0x4(Block_IO) open_flag=0x3(FREAD|FWRITE) device=(192, 4)
PLEX 0 flags=0x0
VE 0 [active]
start=0, end=302076671, (stripe)grp_size=17, stripe_unit_size=256
/dev/dsk/dks6d3s7 (17777424 blks)
/dev/dsk/dks6d4s7 (17777424 blks)
/dev/dsk/dks12d3s7 (17777424 blks)
/dev/dsk/dks12d4s7 (17777424 blks)
/dev/dsk/dks14d3s7 (17777424 blks)
/dev/dsk/dks14d4s7 (17777424 blks)
/dev/dsk/dks20d3s7 (17779016 blks)
/dev/dsk/dks20d4s7 (17779016 blks)
/dev/dsk/dks22d3s7 (17777424 blks)
/dev/dsk/dks22d4s7 (17777424 blks)
/dev/dsk/dks30d4s7 (17769232 blks)
/dev/dsk/dks36d4s7 (17769232 blks)
/dev/dsk/dks38d4s7 (17769232 blks)
```

12-2.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
/dev/dsk/dks44d4s7 (17769232 blks)
/dev/dsk/dks46d4s7 (17769232 blks)
/dev/dsk/dks52d4s7 (17769232 blks)
/dev/dsk/dks54d4s7 (17769232 blks)
```

```
VOL polar_ptmp flags=0x1, [complete] (node=NULL)
DATA flags=0x0() open_flag=0x0() device=(192, 5)
PLEX 0 flags=0x0
VE 0 [active]
start=0, end=17769231, (cat)grp_size=1
/dev/dsk/dks30d3s7 (17769232 blks)
....
```

12-2.c

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Example IRIX read(2) Sequence

```
1 read(2)
  ----- user->kernel-----
2 systrap
3 syscall
  --sysent[]---ml->os-----
4 read
5  _read
  VOP_READ
  ----xfs_vnodeops[]----os->xfs-----
6  xfs_read
7  xfs_read_file
8  chunkread
  VOP_STRATEGY
  ----xfs_vnodeops[]-----
9  xfs_strategy
  (bp->bp_target set to mount mp->m_ddev_targp, which has a pointer
  to the bdevsw entry)
10 xfs_strat_read
11 xfsbdstrat (mp, bp) (use bp->b_target->bdevsw)
12 bdrv [bdstrat macro]

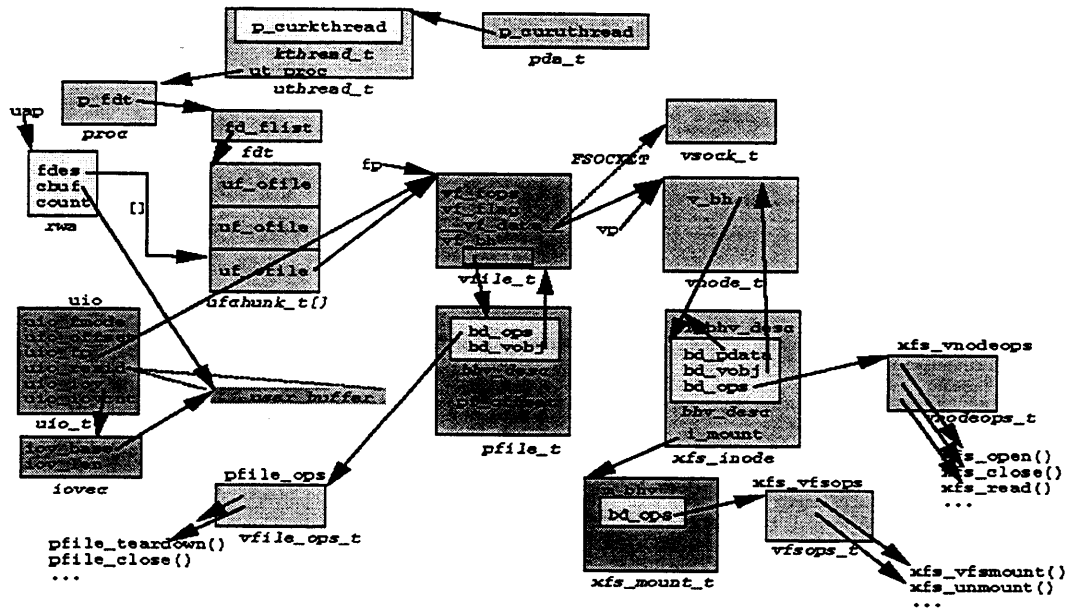
  ----bdevsw[]-----xfs->xlv-----
13 xlvstrategy (minor is index to subvolume)
14 xlv_lower_strategy
15 griostrategy (major 0: use hwgraph)
16 bdrv [bdstrat macro]

  ----bdevsw[]-----xlv->disk driver---
17 dkscstrategy
18 dksccommand
19 dkscstart

  ---lun vertex->target vertex->controller vertex->scicommand
```

```
20 qlcommand
21 ql_entry
22 ql_start_scsi
23 ql_PCI_OUTH moves to registers
```

System Call Layer - Read



read(struct rwa *uap, rval_t *rvp)

12-4

22jul1998

TR-IKI rev 0.7b SGI Proprietary

- user arguments:

```

struct rwa {
    sysarg_t fdes;
    char *cbuf;
    usysarg_t count;
    sysarg_t off64;          /* off64 is the offset for 64 bit apps */
    sysarg_t off1;          /* off1 and off2 is the 64bit offset */
    sysarg_t off2;          /* for 32 bit apps */
};
    
```

- `_read(uap, rvp, readwrite)`

`_read(struct rwa *uap, rval_t *rvp, enum rwrtn wherefrom)`

- `getf((int)uap->fdes, &fp)` set `fp` by finding the `pda's p_curuthread->thread ut_proc->proc p_fdt->fd_list->uf_ofile[fd]`
 - `proc` points to the exandable table of the user's open files
 - user's file descriptor (`fd`) indexes into the table of the user's open files
 - the open file table points to the system table of open files (`vfile` anchors a list of "physical" files - each with its own offset)
 - the open file table points to the system table of open files (`vfile` anchors a list of "physical" files - each with its own offset)
- store user args in a local `uio` structure
 - `uio_resid` = user buffer size
 - there is only one `iovec` for a `read(2)` -- there may be many for a `readv(2)` [similar to *UNICOS listio(2)*]
- if `vfile_t` is flagged as a socket, do a socket receive (`vfile` points to a socket if the file type is `FSOCKET`)
 - for regular files, the `vfile` points to the `vnode`
- set `vnode` pointer "`vp`" to the `vnode_t`
 - the `xfs_inode` describes the location of the data
- set `ioflag` from `vf_flag` if special handling (`FDIRECT`, `FSYNC`, ...)
- set file offset in `uio` struct by calling `pfile_getoffset()` via the `pfile_ops[]` table ("physical" file)
- `VOP_READ(vp, &uio, ioflag, fp->vf_cred, &ut->ut_flid, error);`
 - locates the `vnode's` behavior description; this points to the `xfs_vnodeops[]` for an XFS file
 - this is a table of functions that do operations on the file; in the case of a file on an XFS filesystem it uses the `xfs` `vnode` operations
 - for an XFS file, the `VOP_READ` macro calls `xfs_read()`

12-4.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

○ the vnode points to the inode for this type of filesystem; for a file on an XFS filesystem the vnode points to an xfs_inode

```
>> proc | grep make
a800000047b60c00 1 346986 342801 346986 8827 a800000047b61398 make

>> proc -f a800000047b60c00
      PROC ST  PID  PPID  PGID  UID          WCHAN NAME
=====
a800000047b60c00 1 346986 342801 346986 8827 a800000047b61398 make

SELECTED FIELDS FROM THE KTHREAD STRUCT AT 0xa800000026af2400:
K_FLAGS(0x240020)=KT_SLEEP|KT_HOLD|KT_WSV
K_W2CHAN=0x0,K_STACK=0xffffffffffff8000, K_STACKSIZE=16384
K_PRTN=1, K_PRI=-2, K_BASEPRI=-2, K_SQSELF=0, K_ONRQ=-1
K_SONPROC=-1, K_BINDING=-1, K_MUSTRUN=-1
K_LASTRUN=5, K_CPUSSET=1, K_EFRAME=0x0, K_LINK=0x0
K_INHERIT=0x0, K_INDIRECTWAIT=0x0
K_RFLINK=0xa800000026af2400, K_RBLINK=0xa800000026af2400
K_FLINK=0xa800000026af2400, K_BLINK=0xa800000026af2400

SELECTED FIELDS FROM THE PROC STRUCT:
P_CHILDPIDS=0xa800000266f722c0, P_SLINK=0x0, P_SHADDR=0x0

OPEN FILES FOR PROC 0xa800000047b60c00:

  FD          FILE RCNT          DATA          BH          FLAGS
-----
0  a8000003a38c7a80  14  a800000340040600  a800000366c4e358  3
1  a8000003a38c7a80  14  a800000340040600  a800000366c4e358  3
2  a8000003a38c7a80  14  a800000340040600  a800000366c4e358  3
3  a800000220a666a0   1  a8000003767312000  a8000002241b38d8  1

=====
1 active processes found
```

● The process's open files may also be displayed with the "file" directive:

```
The address of the file may be used:
>> file a800000220a666a0
      FILE RCNT          DATA          BH          FLAGS
-----
a800000220a666a0  1  a8000003767312000  a8000002241b38d8  1
=====
1 file struct found
```

```
The address of a proc may be specified to show all open files:
>> file -p a800000047b60c00
```

```
OPEN FILES FOR PROC 0xa800000047b60c00:

  FD          FILE RCNT          DATA          BH          FLAGS
-----
0  a8000003a38c7a80  14  a800000340040600  a800000366c4e358  3
1  a8000003a38c7a80  14  a800000340040600  a800000366c4e358  3
2  a8000003a38c7a80  14  a800000340040600  a800000366c4e358  3
3  a800000220a666a0   1  a8000003767312000  a8000002241b38d8  1
4 file structs found
```

● FLAGS:

```
#define FREAD      0x01
#define FWRITE     0x02
```

● The FILE addresses point to vfile_t structures: (the first 3 seem to point to stdin/stdout/stderr, so use 4th)

```
>> print *(vfile_t *)a800000220a666a0
struct vfile {
  vf_bh = bhv_head_t {
    bh_first = 0xa8000002241b38d8
  }
  vf_lock = 2
  vf_flag = 1
  vf_count = 1
  vf_msgcount = 0
  vf_data = 0xa8000003767312000
  vf_cred = 0xa8000001801eb780
  vf_cpr = union {
    cu_mate = 0xffffffff00000002
```

```

    cu_ckpt = -1
}
)

```

- The DATA addresses point to vnode_t's:

```

>> print *(vnode_t *)a800003767312000
struct vnode {
  v_list = struct vnlst {
    vl_next = 0xa800003767312000
    vl_prev = 0xa800003767312000
  }
  v_flag = 67108864
  v_count = 6
  v_namecap = 2
  v_listid = 9
  v_intpcount = 0
  v_type = (VDIR=2)
  v_rdev = 0
  v_vfsmountedhere = (nil)
  v_vfsp = 0xa80000002497a80
  v_stream = (nil)
  v_filocks = (nil)
  v_filocksem = mutex_t {
    m_bits = 0
    m_queue = (nil)
  }
  v_number = 2911825
  v_bh = bhv_head_t {
    bh_first = 0xa8000003e1991b30
  }
  v_hashp = 0xa800002221355400
  v_hashn = 0xa800002264b62200
  v_mreg = 0xa800003767312000
  v_mregb = 0xa800003767312000
  v_dbuf = 0
  v_pgcnt = 0
  v_dpages = (nil)
  v_dpages_gen = 0
  v_buf = (nil)
  v_bufgen = 3
  v_buf_lock = mutex_t {
    m_bits = 0

```

12-4.d

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

    m_queue = (nil)
  }
  v_pc = vnode_pcache_t {
    v_pcacheref = 0
    v_pcacheflag = 0
    v_pagecache = pcache_t {
      pc_size = 1
      pc_count = 0
      pc_un = union {
        pc_un_list = (nil)
        pc_un_hash = (nil)
      }
    }
  }
  v_ckpt = (nil)
)

```

- the BH addresses are bhv_desc_t's in the pfile_t's

```

>> print *(bhv_desc_t *)a80000366c4e358
struct bhv_desc {
  bd_pdata = 0xa800000366c4e340
  bd_vobj = 0xa800003a38c7a80
  bd_ops = 0xc0000000142f948
  bd_next = (nil)
}

>> findsym 0xc0000000142f948
=====
0xc0000000142f948 --> pfile_ops
=====
1 symbol found

```

- the v_bh structure in the vnode (see the px output above) points to the inode's bhv_desc_t:

```

>> print *(bhv_desc_t *)0xa8000003e1991b30
struct bhv_desc {
  bd_pdata = 0xa8000003e1991b00 /* this points to the inode containing us */
  bd_vobj = 0xa800003767312000 /* this points to the vnode */
  bd_ops = 0xc0000000143a218

```

12-4.e

22jul1998

TR-IKI rev 0.7b SGI Proprietary


```
    bd_next = (nil)
}
```

- you can tell from its pointer to vnode operations that this is an XFS file:

```
>> findsym 0xc00000000143a218
=====
0xc00000000143a218 --> xfs_vnodeops
=====
1 symbol found

>> print *(xfs_inode_t *)0xa8000003e1991b00
struct xfs_inode {
    i_hash = 0xc000000004cb9400
    i_next = 0xa80000142029b000
    i_prevp = 0xc000000004cb9400
    i_mount = 0xa800000026970800
    i_mnext = 0xa8000000452e0300
    i_mprev = 0xa8000015255ed200
    i_bhv_desc = struct bhv_desc {
        bd_pdata = 0xa8000003e1991b00
        bd_vobj = 0xa800003767312000
        bd_ops = 0xc00000000143a218
        bd_next = (nil)
    }
    i_udquot = (nil)
    i_pdquot = (nil)
    i_ino = 12783736
    i_blkno = 6372464
    i_dev = 50331658 /* 300000a is major 192 (XLV), minor 10 */
    i_len = 16
    i_boffset = 6144
    i_afp = (nil)
    i_df = xfs_ifork_t {
        if_bytes = 16
        if_real_bytes = 0
        if_broot = (nil)
    }
    ....
    i_d = xfs_dinode_core_t {
        di_magic = 18766
        di_mode = 16877

```

```
    di_version = '\001'
    di_format = '\002'
    di_onlink = 3
    di_uid = 8827
    di_gid = 1037
    di_nlink = 3
    di_projid = 0
    di_pad = {
        [0] 0
        [1] 0
        [2] 0
        [3] 0
        [4] 0
        [5] 0
        [6] 0
        [7] 0
        [8] 0
        [9] 0
    }
    di_atime = xfs_timestamp_t {
        t_sec = 888010746
        t_nsec = 978164536
    }
    di_mtime = xfs_timestamp_t {
        t_sec = 888010676
        t_nsec = 342110995
    }
    di_ctime = xfs_timestamp_t {
        t_sec = 888010676
        t_nsec = 342110995
    }
    di_size = 4096
    di_nblocks = 1
    di_extsize = 0
    di_nextents = 1
    di_anextents = 0
    di_forkoff = 0
    di_aformat = '\002'
    di_dmevmask = 0
    di_dmstate = 0
    di_flags = 0
    di_gen = 0
}
}
```

- The vnode may also be displayed with the "vnode" directive:

```
>> vnode a800003767312000 (address from the DATA column of >> file or >> proc -f)
=====
VNODE RCNT TYPE VFSP DEV BH
=====
a800003767312000 6 VDIR a800000002497a80 0 a8000003e1991b30
=====
1 vnode struct found

>> vnode -f a800003767312000
=====
VNODE RCNT TYPE VFSP DEV BH
=====
a800003767312000 6 VDIR a800000002497a80 0 a8000003e1991b30
=====
V_LIST:
VL_NEXT=0xa800003767312000, VL_PREV=0xa800003767312000
V_FLAG=0x4000000, V_NAMECAP=0x2
V_VFSMOUNTEDHERE=0x0, V_STREAM=0x0
V_FILOCKS=0x0, V_FILOCKSEM=0x0
V_NUMBER=2911825, V_LISTID=9, V_INTPCOUNT=0
V_HASHP=0xa800002221355400, V_HASHN=0xa800002264b62200
V_PGCNT=0, V_DBUF=0, V_DPAGES=0x0
V_BUF=0x0, V_BUF_LOCK=0x0, V_BUFGEN=3
=====
1 vnode struct found

VFSP: v_vfsp -- pointer to struct vfs, which is the virtual filesystem (one per
mounted filesystem, the vfs points to the filesystem-dependent mount table. See
"Mounted Filesystems" above.

>> px *(struct vfs *)a800000002497a80
struct vfs {
    vfs_next = 0xa80000000134f680
    ...
    vfs_fstype = 0x1
    ...
    vfs_bh = bhv_head_t {
        bh_first = 0xa800000026970800
    }
}
)
```

12-4.h

22jul1998

TR-IKI rev 0.7b SGI Proprietary

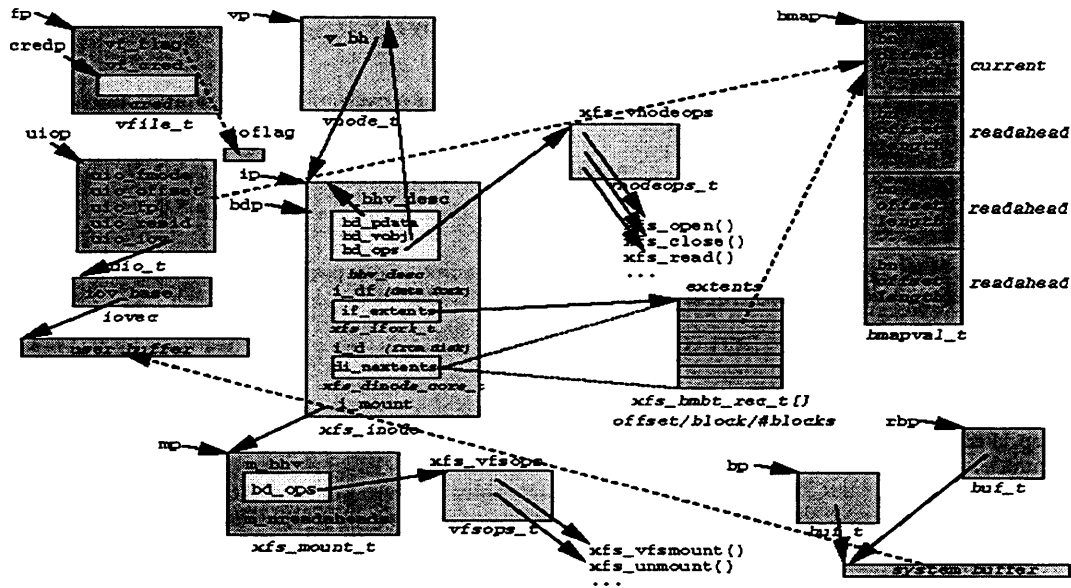
DEV: v_rdev -- device number for special inodes (VCHR, VBLK)
BH: v_bh -- address of the filesystem dependent inode's behavior structure

12-4.i

22jul1998

TR-IKI rev 0.7b SGI Proprietary

(XFS) Filesystem Layer - Read



```
xfs_read(bhv_desc_t *bdp, uio_t *uiop, int ioflag, cred_t *credp, flid_t *fl)
```

- set vp to the vnode (via behavior descriptor argument)

- set ip to the inode
- set mp to the mount structure
- check preferred iosizes (in the uio; IO_UIOSZ)
- for a regular file:
 - if (ioflag & IO_DIRECT) call xfs_diordwr()
 - else (buffer cache I/O): xfs_read_file(bdp, uiop, ioflag, credp)

```
xfs_read_file(bhv_desc_t *bdp, uio_t *uiop, int ioflag, cred_t *credp)
```

- while uio_resid != 0: /* uio_resid is the length of the user's request */
 - nbmaps = m_nreadaheads

```
the filesystem's mount table m_nreadaheads value:
>> print ((xfs_mount_t *)0xa80000026970800)->m_nreadaheads
4
```

- call xfs_iomap_read() to map the request (and read-aheads) into 4 bmapval structures:
 - call xfs_bmap() to create a list of the inode's extents from current offset to end of file
 - set iosize to starting/ending blocks (rounded down to page boundaries)
 - call xfs_next_bmap() to convert the beginning of the request into a bmapval (using the list of extents)
 - bmapval's offset and length returned in disk sector units
 - for each additional bmapval:
 - call xfs_next_bmap() to convert the next part of the request into a bmapval (using the list of extents)
 - return the number of filled-in bmapval's (nbmap) to xfs_read_file's "nbmaps"
- while uio_resid && nbmaps:
 - /* see discussion of system buffers below */
 - call chunkread(vp, bmap, read_bmaps, credp) :
 - call fetch_chunk(vp, bmap, cred) to set up the buf_t and its pages for a bmapval:
 - select page size from policy module
 - call gather_chunk(vp, bmap, &gc_flags, &start_page_size, &end_page_size):

- call bp_insert() to locate or insert a buf_t in the vnode's v_buf tree
 - while bmap_length:
 - find and/or allocate pages to link to the buf_t's b_pages list
 - mark the buf B_DONE or B_PARTIAL (partly filled)
 - return bp
 - return bp
 - if the buf_t is B_DONE, call chunkreada(vp, ++bmap, cred) for each of the following bmapval's
 - bp = fetch_chunk(vp, bmap, cred);
 - VOP_STRATEGY(vp, bp)
 - (for XFS filesystems, jump thru xfs_vnodeops[] to xfs_strategy())
- return bp
- if the buf_t is B_PARTIAL, call patch_chunk() to fill the initialized pages of this buf_t, then chunkreada(vp, ++bmap, cred) for each of the following bmapval's
- return bp
- (buf is not B_DONE or B_PARTIAL)
 - VOP_STRATEGY(vp, bp);
- call chunkreada(vp, ++bmap, cred) for each of the following bmapval's
- sleep in biowait(bp)
- (chunkread) return bp
- if bp->b_resid != 0 mark buf_t B_DONE and break from while loop
- else (bp->b_resid == 0) call biomove()/uiomove() to move any already-buffered data from the buf b_pages list to the user buffer at iov_base; these decrement uiio_resid for each move
- bmapp++; nbmaps--
- continue until all bmapval's are done (nbmaps) or request done (uiio_resid == 0)
- continue until request done (uiio_resid == 0)

12-5.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

xfs_strategy(bhv_desc_t *bdp, buf_t *bp)

- call xfs_strat_read(bdp, bp);

xfs_strat_read(bhv_desc_t *bdp, buf_t *bp)

- call xfs_bmapi() to locate the file's extents
- for each block of the request decide:
 - if a hole in the file, zero out the buffer
 - if not a hole, call getrbuf() to allocate a buf_t (rbp) for the driver initialize the rbp->buf_t from the bp->buf_t
- xfsbdstrat(mp, rbp);
- iowait(rbp)
- iodone(bp);

xfsbdstrat((struct xfs_mount *mp, struct buf *bp)

- my_bdevsw = get_bdevsw(bp->b_edev);
- /* use hwgraph for major 0, or bdevsw[] for XLV major 192 */
- bdstrat(my_bdevsw, bp);
- /* which is a macro for: */
- bdrv(my_bdevsw, DC_STRAT, bp)

bdrv(bdevsw *my_bdevsw, int routine, ...)

- case DC_STRAT: func = (bdevfunc_t)my_bdevsw->d_strategy;
- (*func)(a1, a2, a3, a4, a5);
- /* calls XLV driver xlvstrategy(bp) or disk driver dkscstrategy(bp) */
- display a file's extents:

12-5.c

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

>> print *(xfs_inode_t *)0xa8000003e1991b00 | grep ext
i_next = 0xa80000142029b000
i_mnext = 0xa8000000452e0300
  bd_next = (nil)
  if_ext_max = 9
    if_extents = 0xa8000003e1991ba0 /* the extent list pointer */
    if_inline_ext = {
i_next_offset = 0
i_ext_attr = (nil)
  di_extsize = 0
  di_nextents = 1
  di_anextents = 0

>> dump 0xa8000003e1991ba0 2 /* each extent is 128 bits 55/52/21 bits */
a8000003e1991ba0: 0000000000000000 000001876d200001 |.....m ..

$ interpret extent2 000001876d200001 /* see -mix/irix/tools */

word 2 of an XFS extent

bits 63-21: 011000011101101101001 hex 0x0c3b69
right 43 bits of starting block number

bits 20-0 contain binary 1 decimal 1
number of blocks

a123(542): htd 0x0c3b69 /* file resides in fs block 801,641 */
801641

```

System Buffers

buffers

- page cache
 - all pages of physical memory
 - indexed by pfdat
 - shared by virtual memory system and file cache
 - if nothing but file I/O is going on almost all of the page cache will become dedicated to caching file pages
 - a file's pages may be mapped into a user's address space with mmap(2)
- chunk cache
 - the variable-length buffers headed by buf_t structures
 - used for file I/O
 - groups of pages, indexed by file
- monitor the system buffers with bufview(1)
- system buffer cache is anchored in global_buf_hash
- the device list is anchored in global_dev_hash
- number of buf_t's:

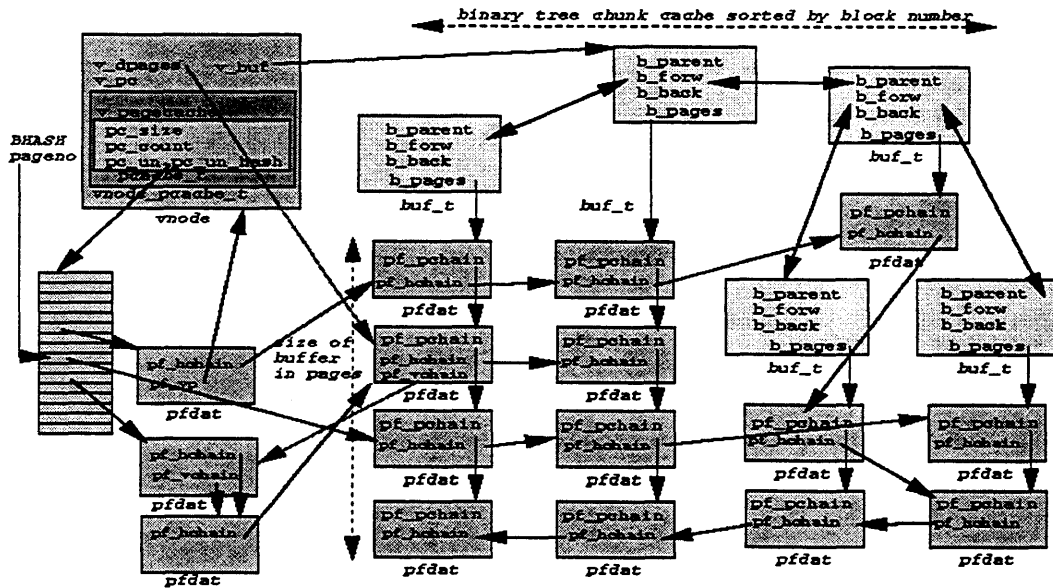

```

>> whatis v
0xc0000000014b8380

>> print *(struct var *)0xc0000000014b8380 | grep buf
v_buf = 125000
v_hbuf = 8192

```
- ngetblkdev(dev_t dev, size_t len) gets a free buf from the bfreelist[]
 - buf is chained onto the device hash list
 - buf is assigned a buffer (may re-use memory from the fraglist[])

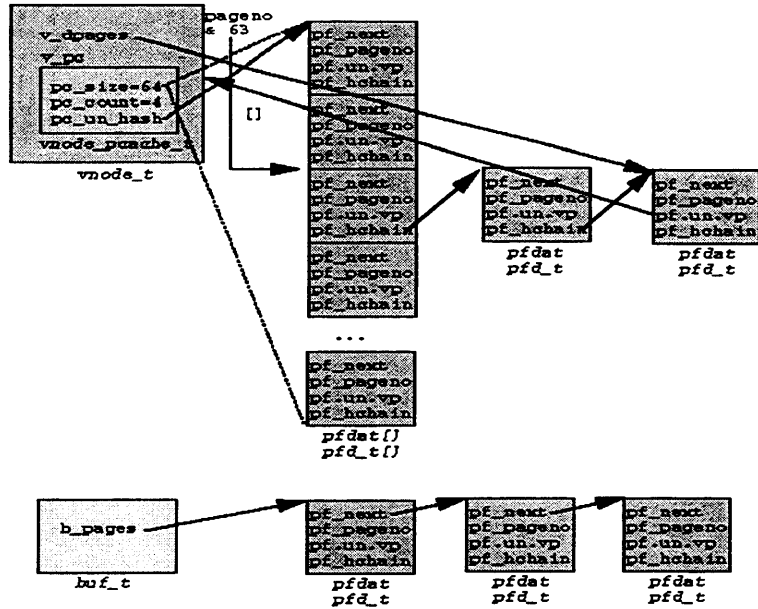
Both the chunk cache and page cache for a file are anchored in the vnode:



- the buf's are sorted into a tree
 - the tree structure is balanced using b_balance (see bp_balance())
 - there is a delwri (delayed write) chain using b_dforw (and b_dback)

- the buffers consist of a list of pages, represented by pfdat's
 - fetch_chunk()->gather_chunk() locates a buf_t and gathers pages to one buf's bp_pages list to represent one file extent
 - it allocates buffers where none exist already
- all pages representing data from the file are linked to vnode via a hash table (or directly, if a small number)
 - all such pfdat's point back to the vnode (pf_vp)
 - any pfdat not associated with a file is "anonymous" with pf_tag pointing to an "anon" structure (pf_vp/pf_tag is a union); anon structures connect the page with space on the swap device
 - pfind() aka vnode_pfind() -> vnode_pfind_nolock() -> pcache_find() -> pcache_search() to search for a vnode's page in a hash list
- all dirty (written-to) pages are on the v_dpages list
- the "bdflush" service thread flushes dirty pages from the chunk cache
- the "pdflush" service thread flushes dirty pages from the page cache
- the "vhand" service thread frees up unused pages to a freelist
 - when pf_use == 0 the page can go onto the free list
- the "coalesced" service thread frees up physically contiguous pages to combine as large pages

detail on the vnode's page hash list:



```
To view a hash table attached to a vnode:
>> print (*(vnode_t *)0xa800002626770900)-->v_pc
<<< notice pc_un_hash: the hash table and pc_size: #elements in hash table >>>
>> whatis -l pfdat
```

```
<<< displays the pfdat structure length >>>
>> sh
$ parray -o printarray.out pfdat 64 64 <<<hash table address>>> | cat > printarray.dirs
<<< find this tool in -mix/irix/tools >>>
$ exit
>> from printarray.dirs
<<< this reads/executes the directives, which write their output to printarray.out >>>
>> sh
$ grep ^struct printarray.out
64 <<< number of structures printed >>>
$ grep pf_hchain printarray.out
<<< shows all the heads pointed to by the hash list >>>
$ exit

/* to print members of such a linked list: */
>> walk -s pfdat pf_hchain <<<<<first element in a list>>>>
```

- there are 32 disk blocks (sectors) in a 16KB page



- Detail of the buf structure:

```
typedef struct buf {
/*
 * These first 4 fields must match the hbuf definition
 * below. DO NOT CHANGE THEM OR REARRANGE THEM.
 */
sema_t b_lock; /* lock for buffer usage */
uint64_t b_flags; /* see defines below */
struct buf *b_forw; /* headed by d_tab of conf.c */
struct buf *b_back; /* " */
struct buf *bd_forw; /* device based hash chain */
struct buf *bd_back; /* " */
struct dhbuf *bd_hash; /* hash bucket head */
struct buf *av_forw; /* position on free list, */
struct buf *av_back; /* if not BUSY */
dev_t b_edev; /* major+minor device name */
int b_error; /* returned after I/O */
off_t b_offset; /* vnode offset (in basic blocks) */
buftarg_t *b_target; /* route to I/O device */
};
```

```

unsigned b_bcount;          /* transfer count */
unsigned b_resid;          /* words not transferred after error */
unsigned b_remain;        /* virt b_bcount for PAGEIO use only */
unsigned b_bufsize;       /* size in bytes of allocated buffer */
__unsignd_t b_sort;       /* key with which to sort on queue */
union {
    caddr_t b_addr;        /* low order core address */
    int *b_words;         /* words for clearing */
    struct pfdat *b_pfdat; /* pointer into b_pages list */
    daddr_t *b_daddr;     /* disk blocks */
} b_un;
struct vnode *b_vp;       /* object associated with bp */
daddr_t b_blkno;          /* block # on device */
clock_t b_start;         /* request start time */
struct pfdat *b_pages;   /* page list for PAGEIO */
void *b_alenlist;        /* address, length lists */
void (*b_relse)(struct buf *); /* function called by brelse */
sema_t b_iodonesema;     /* lock for waiting on I/O done */
void (*b_iodone)(struct buf *); /* function called by iodone */
int (*b_bdstrat)(struct buf *); /* function called by bwrite */
void *b_private;         /* for driver's use */
void *b_fsprivate;       /* private ptr for file systems */
void *b_fsprivate2;      /* private ptr for file systems */
void *b_fsprivate3;      /* private ptr for file systems */
short b_pincount;        /* count of times buf is pinned */
ushort b_pin_waiter;     /* someone waiting for unpin? */
char b_ref;              /* # of free trips through freelist */
char b_balance;          /* tree balance */
char b_listid;           /* free list number */
char b_bvtype;           /* bufview flags */
struct buf *b_dforw;     /* vnode delwri chain */
struct buf *b_dback;     /* vnode delwri chain */
struct buf *b_parent;    /* hash parent pointer */
void *b_grio_private;    /* private data for grio */
struct buf *b_grio_list; /* list of bps used in grio */
#ifdef DEBUG_BUFTRACE
    struct ktrace *b_trace; /* per buffer trace buffer */
#endif
} buf_t;

```

- Detail of the pfdat structure:

12-7.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

typedef struct pfdat {
    struct pfdat *pf_next; /* Next free pfdat. */
    union {
        struct pfdat *prev; /* Previous free pfdat. */
        sm_swaphandle_t swphdl; /* Swap hdl for anon pages */
    } p_swpun;
#ifdef _VCE_AVOIDANCE
    int pf_vcolor:8; /* Virtual cache color. */
    /* bit 0: PE_RAWWAIT rawwait */
    /* bits 1..7: PE_COLOR */
    pf_flags:24; /* Regular page flags */
#else
    uint pf_flags; /* Regular & NUMA page flags */
#endif
    ushort_t pf_use; /* Share use count. */
    unsigned short pf_rawcnt; /* Count of processes */
    /* doing raw I/O to page*/
    unsigned long pf_pageno; /* Object page number */
    union {
        struct vnode *vp; /* Page's incore vnode. */
        void *tag; /* Generic hash tag. */
    } p_un;
    struct pfdat *pf_hchain; /* Hash chain link */
    union pde *pf_pdepl; /* Primary pde ptr */
    union {
        struct rmap *pf_revmap; /* Reverse map pointer */
        union pde *pf_pdeptr; /* Page tbl entry ptr */
    }
#ifdef CELL
    time_t pf_utimestamp; /* timestamp for zero use cnt */
#endif
    } p_rmapun;
#ifdef MULTIKERNEL
    __uint64_t pf_exported_to; /* bitstring of cells page is currently export to */
#endif
} pfd_t, pfde_t;

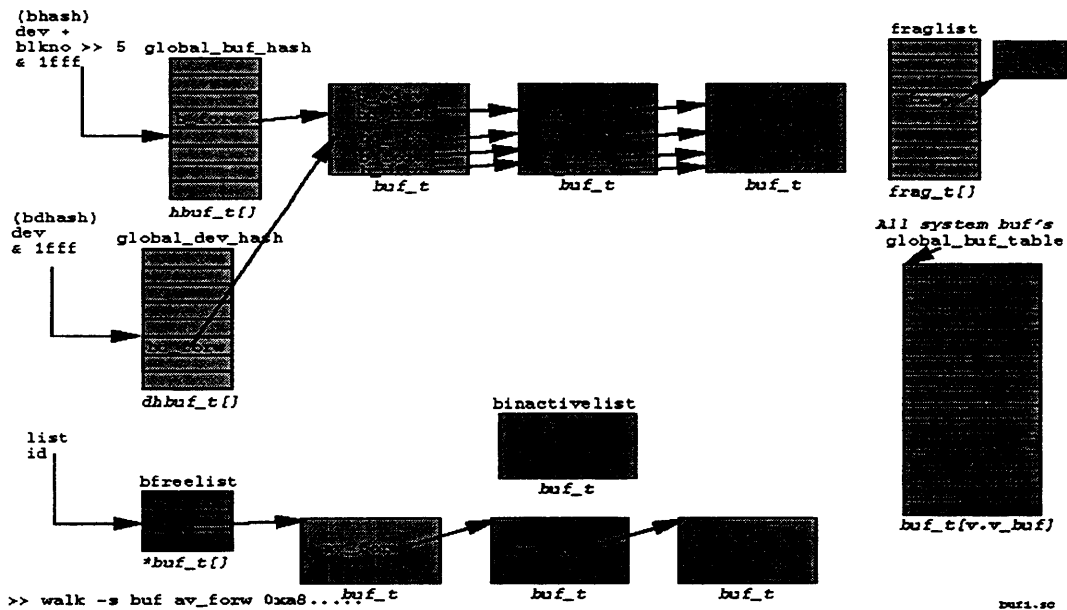
```

- There is a "buffer cache" used for filesystem metadata
 - these buf's are indexed by device rather than file
 - see calls to get_buf()
- Some control structures of the "buffer" cache

12-7.c

22jul1998

TR-IKI rev 0.7b SGI Proprietary



Example IRIX write(2) Sequence

```

1 write(2)
----- user->kernel-----
2 systrap
3 syscall
  --sysent[]---ml->os-----
4 write
  _write
  VOP_WRITE
  ----xfs_vnodeops[]----os->xfs---
6 xfs_write
7 xfs_write_file
8 getchunk (or chunkread)
9 biomove
  bdwrite

clock
second_thread
10 bdflush
11 clusterwrite
  VOP_STRATEGY
  ----xfs_vnodeops[]----
12 xfs_strategy (bp_target set to mount mp->m_ddev_targp)
  queue to tail of xfsd_list or
  xfs_strat_write
13 xfs_bmap
14 xfsbdstrat (mp, bp) (use bp->b_target->bdevsw)
  bdstrat [macro]
  bdrv
15
  ----bdevsw[]-----xfs->xlv---
16 xlvstrategy (minor is index to subvolume)
17 xlv_lower_strategy
18 griostrategy (major 0: use hwgraph)
19 bdrv [bdstrat macro]

  ----bdevsw[]-----xlv->disk driver---
20 dkscstrategy
21 dksccommand

```

```

--lun vertex->target vertex->controller vertex->scicommand
23          qlcommand
24          ql_entry
25          ql_start_scsi
26          ql_PCI_OUTH moves to registers

```

write(struct rwa *uap, rval_t *rvp)

- user arguments:

```

struct rwa {
    sysarg_t fdes;
    char *cbuf;
    usysarg_t count;
    sysarg_t off64;          /* off64 is the offset for 64 bit apps */
    sysarg_t off1;          /* off1 and off2 is the 64bit offset */
    sysarg_t off2;          /* for 32 bit apps */
};

```

- _write(uap, rvp, readwrite)

_write(struct rwa *uap, rval_t *rvp, enum rwrtn wherefrom)

- getf((int)uap->fdes, &fp) set fp by finding the pda's p_curuthread->thread ut_proc->proc p_fdt->fd_list->uf_ofile[fd]
 - proc points to the exandable table of the user's open files
 - user's file descriptor (fd) indexes into the table of the user's open files
 - the open file table points to the system table of open files (vfile anchors a list of "physical" files - each with its own offset)
 - the open file table points to the system table of open files (vfile anchors a list of "physical" files - each with its own offset)
- store user args in a local uio structure
 - uio_resid = user buffer size
 - there is only one iovec for a read(2) -- there may be many for a readv(2) [similar to UNICOS listio(2)]
- if vfile_t is flagged as a socket, do a socket send (vfile points to a socket if the file type is FSOCKET)
 - for regular files, the vfile points to the vnode
- set vnode pointer "vp" to the vnode_t

12-8.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

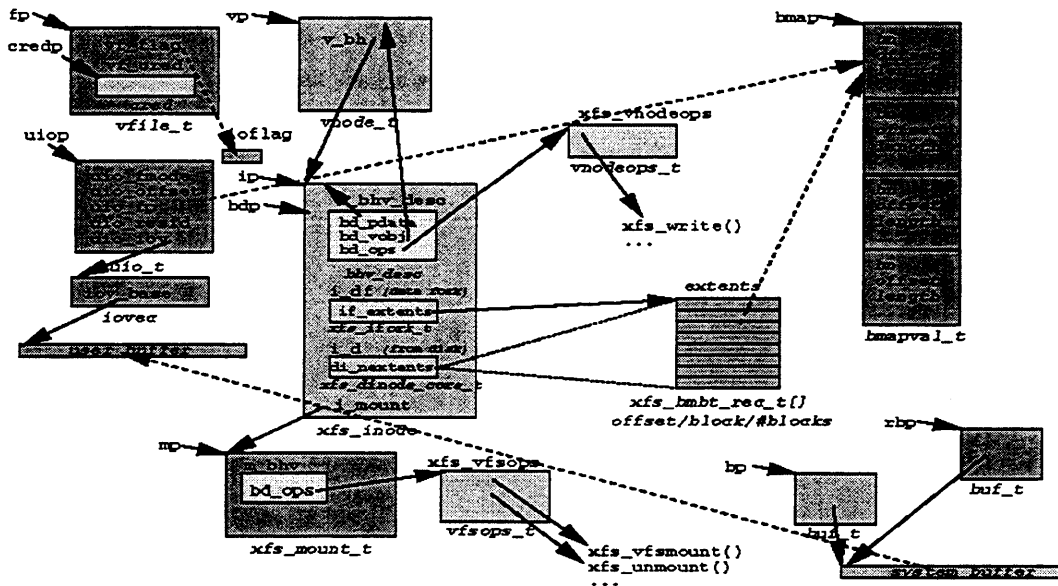
- the xfs_inode describes the location of the data
- set ioflag from vf_flag if special handling (FAPPEND, FSYNC, ...)
- set file offset in uio struct by using VFILE_GETOFFSET_LOCKED
- if IO_DIRECT, call xfs_diordwr()
- for buffered I/O: VOP_WRITE(vp, &uio, ioflag, fp->vf_cred, &ut->ut_flid, error);
 - locates the vnode's behavior description; this points to the xfs_vnodeops[] for an XFS file
 - for an XFS file, the VOP_READ macro calls xfs_write()
 - the vnode points to the inode for this type of filesystem; for a file on an XFS filesystem the vnode points to an xfs_inode

12-8.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

(XFS) Filesystem Layer - Write



```
xfs_write(bhv_desc_t *bdp, uiop_t *uiop, int ioflag, cred_t *credp, flid_t *fl)
```

- set vp to the vnode (via behavior descriptor argument)

- set ip to the inode
- set mp to the mount structure
- if append mode, uio_offset = file size
- check preferred iosizes (in the uio; IO_UIOSZ)
- for a regular file:
 - if (ioflag & IO_DIRECT) call xfs_diordwr()
 - else (buffer cache I/O): xfs_write_file(bdp, uiop, ioflag, credp, &commit_lsn)

```
xfs_wite_file(bhv_desc_t *bdp, uiop_t *uiop, int ioflag, cred_t *credp, xfs_lsn_t *commit_lsn_p)
```

- while uio_resid != 0: /* uio_resid is the length of the user's request */
 - call xfs_build_gap_list to build a list of gaps we are filling (in case a reader is about to read them)
 - call xfs_iomap_write to map the request into bmapval structures representing each extent being written to
 - calls xfs_write_bmap to form the bmap
 - returns the number of bmapval's it filled in
 - for each of the bmapval's formed above:
 - **getchunk(vp, bmapval, credp)** to get the buf
 - use chunkread for special cases
 - biomove(bp, bmapval->pboff, bmapval->pboff, UIO_WRITE, uiop) to move user data into the buffer
 - decrement uio_resid
 - adjust the "gap list" with xfs_delete_gap_list
 - mark the buf for delayed write (**bdwrite(bp)**)
 - use bwrite(bp) for sync. writes
- continue while loop on user request (uio_resid length)

```
getchunk(vnode_t *vp, bmapval_t *bmap, cred_t *cred)
```

- call delalloc_reserve() to ensure that too many delayed allocations are not pending
 - delallocleft is a global counter of how many such allocations may still be made (out of the max. that may exist simultaneously) in the system buffers

- call clusterwrite to flush some out if delallocleft is zero
 - call fetch_chunk(vp, bmap, cred)
 - call gather_chunk (vnode_t *vp, bmapval_t *bmap, int *outflagsp, size_t *start_page_sizep, size_t *end_page_sizep)
 - call bp_insert (vnode_t *vp, struct bmapval *bmap)
 - find the buffer matching the bmap; look in vp->v_buf b_forw/b_back list
 - may call ngetblkdev to get a free one for the list
 - bp->b_offset = bmap->offset
 - collect all pages assoc. with this buf and link to b_pages list
 - start reads as necessary (call cread, which uses VOP_STRATEGY)
 - release(bhead, btail, bp) each buf
 - biowait for outstanding I/O
- return (bp)

bdwrite(bp)

- mark the buffer for delayed write:
bp->b_flags |= B_DELWRI | B_DONE

Control returns to the user. His write to the system buffers is complete.

(or -- buf could "float" to top of freelist)

clock

- every second, wakeup service thread second_thread by releasing second_sema

second_thread

- every bdflushcnt seconds, wakeup bdflush by releasing semaphore bdwakeup (bdflushcnt is 1, unless changed with a syssgi(2))

12-9.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

bdflush

- walks thru nbuf/bdflushr entries of global_buf_table[] (bdflushr is a system tuneable)
- write to disk by calling clusterwrite(bp, age) (if attached to a vnode withb_vp) (call bwrite(bp) if not attached)
- sleeps by psema(&bdwakeup, PZERO)

clusterwrite (buf_t *bp, clock_t start)

- works on vp = bp->b_vp; vnode's list
- works on b_forw/b_back/b_parent list
- call getrbuf to get extra buf's for writes
- write via VOP_STRATEGY(clusterbp->b_vp, clusterbp);
- biowait(clusterbp);
- brelse(clusterbp);

VOP_STRATEGY

xfs_strategy(bhv_desc_t *bdp, buf_t *bp)

- call xfs_strat_write(bdp, bp);

xfs_strat_write(bhv_desc_t *bdp, buf_t *bp)

- for each extent of the request decide:
 - transaction to allocate storage
 - allocate (xfs_bmap())
 - write with xfsbdstrat(mp, rbp);
 - iowait(rbp)
- biodone(bp)

12-9.c

22jul1998

TR-IKI rev 0.7b SGI Proprietary

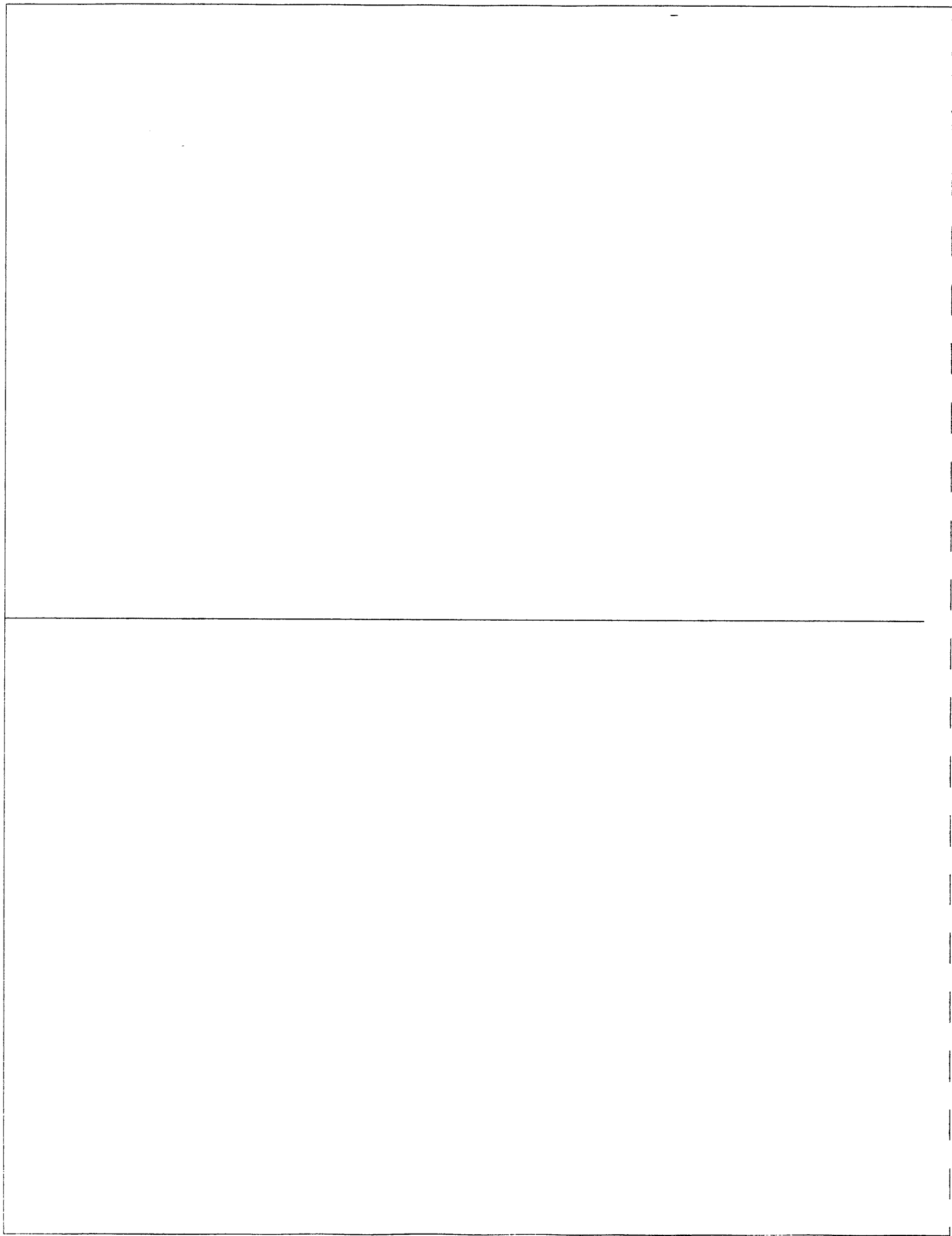
○ brelse()

xfsbdstrat((struct xfs_mount *mp, struct buf *bp)

- my_bdevsw = get_bdevsw(bp->b_edev);
/* use hwgraph for major 0, or bdevsw[] for XLV major 192 */
- bdstrat(my_bdevsw, bp);
/* which is a macro for: */
bdrv(my_bdevsw,DC_STRAT,bp)

bdrv(bdevsw *my_bdevsw, int routine, ...)

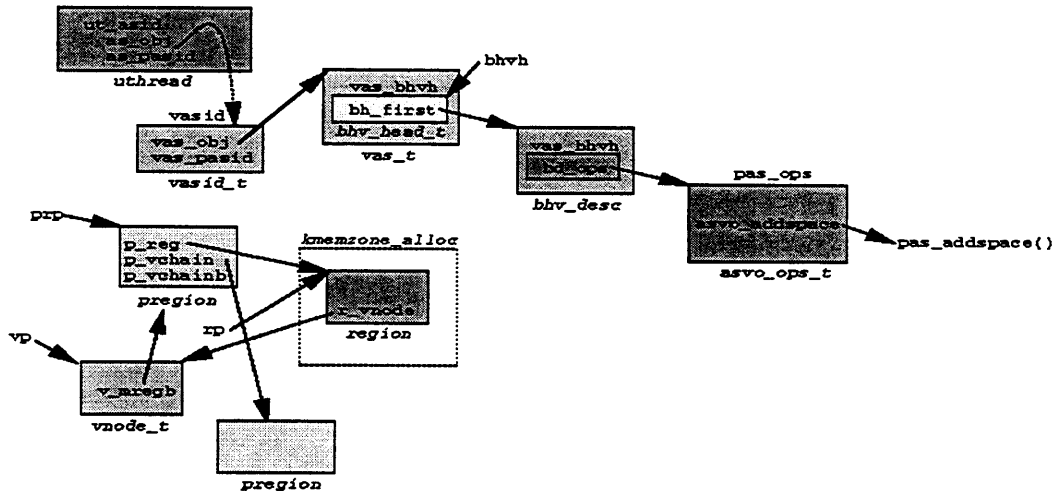
- case DC_STRAT: func = (bdevfunc_t)my_bdevsw->d_strategy;
- (*func)(a1, a2, a3, a4, a5);
/* calls XLV driver xlvstrategy(bp) or disk driver dkscstrategy(bp) */



Module 13: XFS File Management

Reference:

mmap(2) - Memory Mapping a File



```
mmap64() or mmap(0, size, PROT_READ, MAP_SHARED, fd, 0)
                addr length prot flags fd off)
```

- call mmap_common()

mmap_common()

- follow the fd to the vnode (vp)
- set isspec if VCHR or VBLK (character/block special)

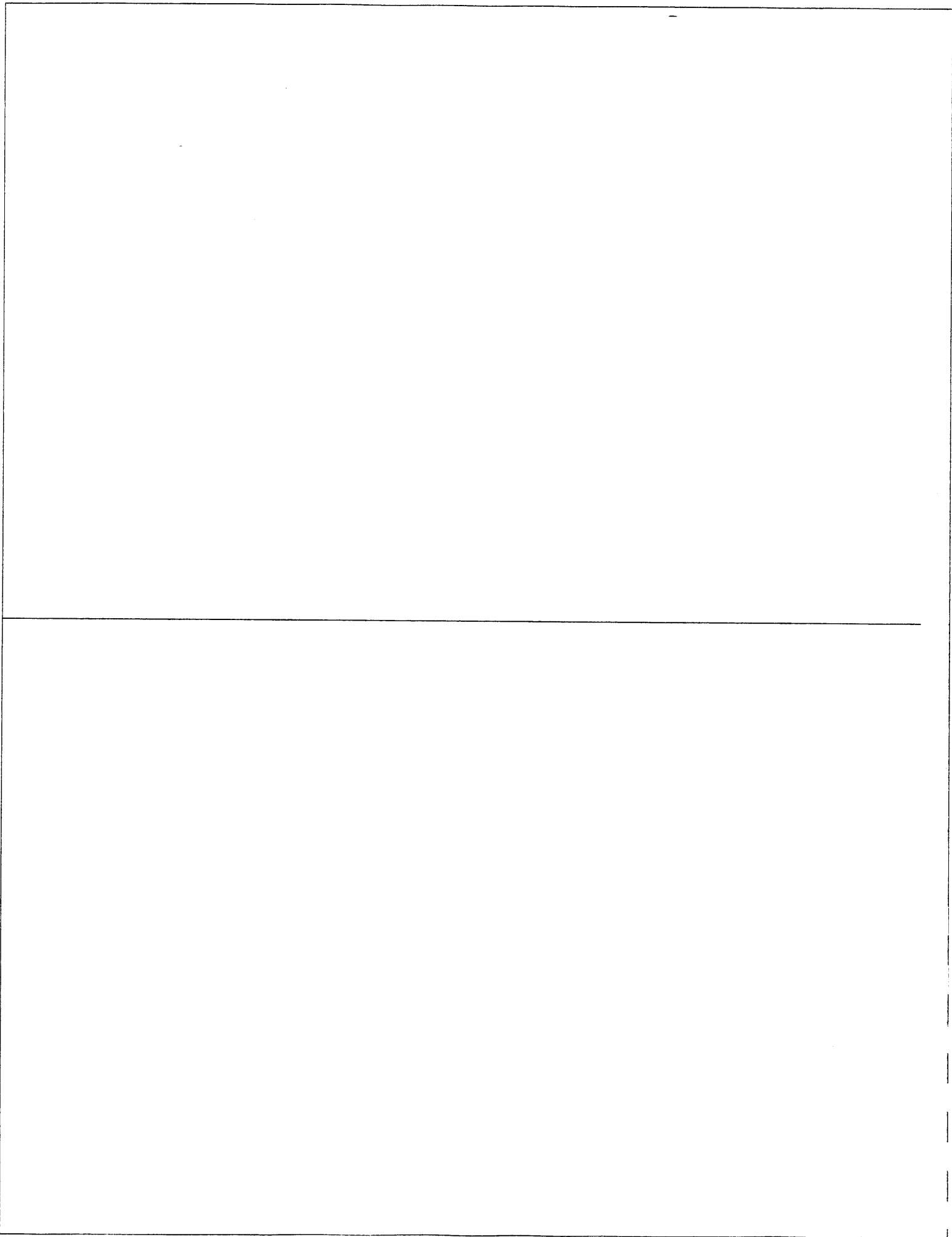
- set up arguments in as_addspace_t structure "asadd"
- asadd.as_op = isspec ? AS_ADD_MMAPDEV : AS_ADD_MMAP;
- as_lookup_current() /* fill in vasid from current uthread */
- VAS_ADDSPACE(vasid, &asadd, &asres) /* macro; see as.h */
 - calls pas_addspace(bhvh->bh_first, vasid.vas_pasid, &asadd, &asres)

pas_addspace (*bhv, pasid, *arg, *asres)

- pas_t *pas = BHV_TO_PAS(bhv) /*uses bd_pdata */
- case (arg->as_op) AS_ADD_MMAP: /* or AS_ADD_MMAP_DEV if file type VCHR/VBLK)
 - pas_addmmap(pas, ppas, arg, &asres->as_addr)

pas_addmmap(pas_t *pas, ppas_t *ppas, as_addspace_t *arg, uvaddr_t *ataddr)

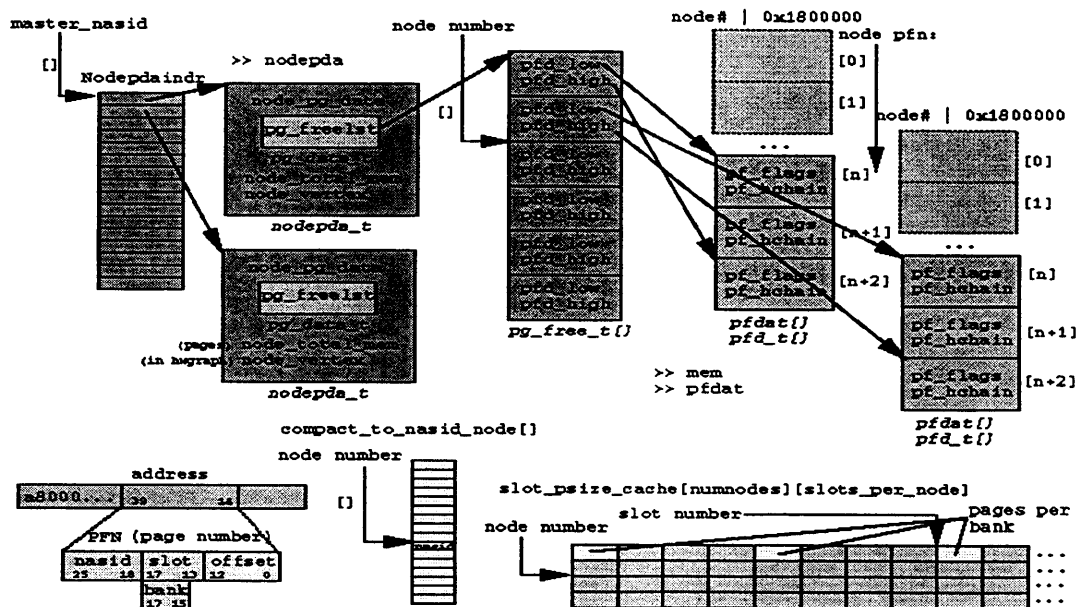
- vgrowreg()
- rp = allocreg(vp, ...) /* allocate a region structure */
- vattachreg(rp,...) /* attach the region to process address space */
 - prp = allocpreg() /* adds a pregon */
 - PREG_INSERT macro calls avl_insert() to insert pregon into its tree
 - attach to pmap
 - mapattach(prp, vp); /* attach pregon to vnode v_mregb chain */



Module 14: pfdats

Reference

pfdat's



- Each node has a list of pfdat's
 - the pfdat's for a node are physically located in each node's own memory
 - each pfdat's PFN is implied by its offset from a well known base of 0x1800000 (NODE_PFDAT_OFFSET); a pfdat structure is

```

64 bytes (0x40) in length
  ■ given the address of a pfdat:
    PFN = address bits 39-32 (i.e. node number) << 18 || ((address bits 31 thru 0 - 0x1800000)/0x40)
  ■ given a PFN:
    address of its pfdat = a800000000000000 || PFN bits 25-18 << 32 (i.e. nasid) || (PFN bits 17-0 * 0x40) + 0x1800000
○ each pfdat contains information about the state and use of that page
○ the list of in-use pfdat anchors can be found by locating the master node's nodepda_t and following it to the array of pg_free_t's.
  Then use any node number (nasid) as an index into the array. (This is how the icrash "pfdat -a" directive lists all pfdat's.)

>> dump master_nasid

c000000001450008: 0000000000000000

>> dump Nodepdaindr 4

c0000000014bc180: a8000000006d9448 a8000001003f4000 |.....m.H.....?#.
c0000000014bc190: a8000002003f4000 a8000003003f4000 |.....?#......?#.

>> px *(nodepda_t *)a8000000006d9448 | grep pg_freelst
pg_freelst = 0xa80000000070f500
pg_freelst_lock = 0x2

or use the ">> nodepda" directive
>> nodepda -f | pg
NODE          NODEPDA
=====
0 a8000000006d9448

NODE_PG_DATA:

PG_FREELIST=0xa80000000070f500
NODE_FREEMEM=11637, NODE_FUTURE_FREEMEM=43018
NODE_EMPTYMEM=11087, NODE_TOTAL_MEM=47932
....

Example: if you were looking for node 2's pfdat's:
>> whatis -l pg_free
NAME          SIZE
=====
pg_free      136

```

```

=====
>> px 0xa80000000070f500+(2*136)
0xa80000000070f610

>> px *(struct pg_free *)0xa80000000070f610
struct pg_free {
  phead = {
    ...
  }
  hiwat = {
    [0] 0x7825
    ...
  }
  pheadend = {
    [0] 0xa8000002004222b8
    ...
  }
  pfd_low = 0xa800000201804240
  pfd_high = 0xa800000201e7ffc0
}
>> pfdat 0xa800000201804240 0xa800000201e7ffc0
=====
PFDAT USE   FLAGS  PGNO  VP/TAG      HASH      PFN
=====
a800000201804240 1    2800   0      0          0    524553
a800000201e7ffc0 0    2001   0      0          0    630783
=====
2 pfdat structs found

```

- The page numbers in a node are not consecutive because there are "gaps" between memory banks of a node
- To display the populated memory banks in each node, dump the "slot_psize_cache" array
 - each entry in the array is a 16-bit integer
 - there are "numnode" rows (major groups); each row represents a node
 - there are "slots_per_node" columns in a row; each column represents a memory "slot"
 - on an ORIGIN, every 4th "slot" contains the number of pages in the bank

```

>> dump numnodes

c00000000145cca4: 00000040          0x40, or 64 nodes

>> dump slots_per_node

c000000001450040: 0000002000001000 0x20, or 32 possible "slots"/node

```

Every fourth "slot" represents bank of memory, therefore "slot"/4 -> bank. This "slot" is distinct from the hardware DIMM slots. On the Origin there are 16 DIMM (Dual In-Line Memory Module) slots on a node board (slots MMXL0-7 and MMXH0-7). MMXL0/MMXH0 are bank 0, MMXL1/MMXH1 are bank 1, etc., for a total of 8 banks. So take the "slot" number from the PFN and divide by 4. The result is a bank number, which is a pair of hardware DIMM's.

in memsupport.c:

```
#ifdef SNO
size = (__int64_t)banks->membnk_bnksz[slot/4];
>> dump slot_psize_cache 20

      node 0      bank 0      bank 1
c00000000145e280: 20000000000000000000 20000000000000000000 | .....
                  bank 2      bank 3
                  slot 8      slot 12
c00000000145e290: 20000000000000000000 20000000000000000000 | .....
                  bank 4      bank 5
                  slot 16     slot 20
c00000000145e2a0: 10000000000000000000 10000000000000000000 | .....
                  bank 6      bank 7
                  slot 24     slot 28
c00000000145e2b0: 10000000000000000000 10000000000000000000 | .....
      node 1
c00000000145e2c0: 20000000000000000000 20000000000000000000 | .....
c00000000145e2d0: 20000000000000000000 20000000000000000000 | .....
c00000000145e2e0: 10000000000000000000 10000000000000000000 | .....
c00000000145e2f0: 10000000000000000000 10000000000000000000 | .....
c00000000145e300: 20000000000000000000 20000000000000000000 | .....
c00000000145e310: 20000000000000000000 20000000000000000000 | .....
```

The total page size for node 0 is c000 pages (if you add up all the slots).

```
o page size:
>> px *(struct icrashdef_s *)&icrashdef | grep page

i_pagesz = 0x4000
i_mapped_kern_page_size = 0x1000000
```

c000*4000=30000000, decimal 805,306,368

The icrash "mem" directive agrees that there is 768 MB of memory in node 0:

```
>> mem
NODE MEMORY:
MODULE SLOT      NODEID NASID MEM_SIZE ENABLED
=====
1 n1             0 0 768 Y
1 n2             1 1 768 Y
...
```

actual vale of 768MB:

```
>> print 768*1024*1024
805306368
```

Node 0 has all 16 physical slots populated (it takes 2 hardware slots to make a bank, and we have 8 banks):

```
$ hinv -c memory -v > mem
...
Memory at Module 1/Slot 1: 768 MB (enabled)
Bank 0 contains 128 MB (Premium) DIMMS (enabled) /* 0x2000 pages */
Bank 1 contains 128 MB (Premium) DIMMS (enabled)
Bank 2 contains 128 MB (Premium) DIMMS (enabled)
Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Bank 4 contains 64 MB (Premium) DIMMS (enabled) /* x1000 pages */
Bank 5 contains 64 MB (Premium) DIMMS (enabled)
Bank 6 contains 64 MB (Premium) DIMMS (enabled)
Bank 7 contains 64 MB (Premium) DIMMS (enabled)
```

- Position of the "slot" number in an address
 - "dump slot_shift" shows that the slot number is bit 27 and up (left) in a kseg0 address


```
>> dump slot_shift
c000000001450030: 0000001b00000000
```
 - "dump slot_bitmask" shows the size of the slot field in an address (i.e. 5 bits)

```
>> dump slot_bitmask
c000000001450038: 000000000000001f
```

● Relationship of node number and nasid:

- "nasid_shift" shows position of nasid (node number) in an address:

```
>> dump nasid_shift
c000000001450020: 0000002000000020      0x20, or 32 bits from the right
```

- "nasid_bitmask" shows the size of the nasid

```
>> dump nasid_bitmask
c000000001450028: 00000000000000ff      nasid is 8 bits
```

- the "compact_to_nasid_node" array shows that node numbers are the same as nasid's:

```
>> dump compact_to_nasid_node 16
c00000000145d810: 0000000100020003  0004000500060007  |.....
c00000000145d820: 00080009000a000b  000c000d000e000f  |.....
c00000000145d830: 0010001100120013  0014001500160017  |.....
c00000000145d840: 00180019001a001b  001c001d001e001f  |.....
c00000000145d850: 0020002100220023  0024002500260027  |.!.#.$.&.'
c00000000145d860: 00280029002a002b  002c002d002e002f  |(.)*+,-./
c00000000145d870: 0030003100320033  0034003500360037  |.0.1.2.3.4.5.6.7
c00000000145d880: 00380039003a003b  003c003d003e003f  |.8.9...;<.=.>?
```

● Address to PFN:

- PFN is bits 39-14 of a physical address (i.e. a80000....)
- divide an address by 0x4000 to shift right 14

● Node and bank to PFN:

Given a node and memory bank, form PFN as node (or nasid) << 18 | bank << 15.

In node 0, the PFN's would be:

```
bank 0: ---0 << 18 | 0 << 15 -- 0x2000 pages --- 0-1fff ---- decimal 0-8191
bank 1: ---0 << 18 | 1 << 15 -- 0x2000 pages --- 8000-9fff ---- decimal 32768-40959
bank 2: ---0 << 18 | 2 << 15 -- 0x2000 pages -- 10000-11fff --- decimal 65536-73727
...
bank 7: ---0 << 18 | 7 << 15 -- 0x1000 pages -- 38000-38fff --- decimal 229376-335871
```

In node 1, the PFN's would be:

```
bank 0: ---1 << 18 | 0 << 15 -- 0x2000 pages -- 40000-41fff --- decimal 262144-270335
bank 1: ---1 << 18 | 1 << 15 -- 0x2000 pages -- 48000-49fff --- decimal 294912-303103
...
```

Compare this with >> pfdat -a output:

PFDAT	USE	FLAGS	PGNO	VP/TAG	HASH	PFN
a8000000018071c0	1	2800	0	0	0	455
a800000001807200	1	2800	0	0	0	456
...						
a80000000187ff80	1	210c	74212	a8000000270447e0	a800000102633f80	8190
a80000000187ffc0	1	20002800	0	0	0	8191
a800000001a00000	0	6001	23960	0	0	32768
a800000001a00040	0	1a000	4817	0	0	32769
...						
a800000001a7ff80	0	2009	242	a800000a80487b00	0	40958
a800000001a7ffc0	1	2800	0	a800000027ffc000	0	40959
a800000001c00000	1	210c	86758	a8000000270447e0	a800000102228540	65536
a800000001c00040	1	2800	0	a800000040004000	0	65537
...						
a80000000243ff80	0	2009	1143	a8000013611ecc00	0	200702
a80000000243ffc0	1	210c	94437	a8000000270447e0	a800000002617fc0	200703
a800000002600000	1	210c	74982	a8000000270447e0	a800000101838000	229376
a800000002600040	1	210c	0	a800000027c4ea60	a800000002217f40	229377
...						
a80000000263ff80	1	210c	95716	a8000000270447e0	a800000001c67f80	233470
a80000000263ffc0	1	2800	0	0	0	233471
a800000101804380	1	210c	61605	a8000000270447e0	a800000102606fc0	262414
a8000001018043c0	2	c	0	a800000002492100	0	262415
...						
a80000010187ff80	1	210c	52708	a8000000270447e0	a80000010202ff80	270334

```

a80000010187ffc0 1 210c 57829 a8000000270447e0 a800000101e33fc0 270335
a800000101a00000 1 210c 37094 a8000000270447e0 a800002101c44000 294912
a800000101a00040 1 210c 38119 a8000000270447e0 a800002101c08040 294913
...

```

Here's another icrash option (but it is extremely slow):

```

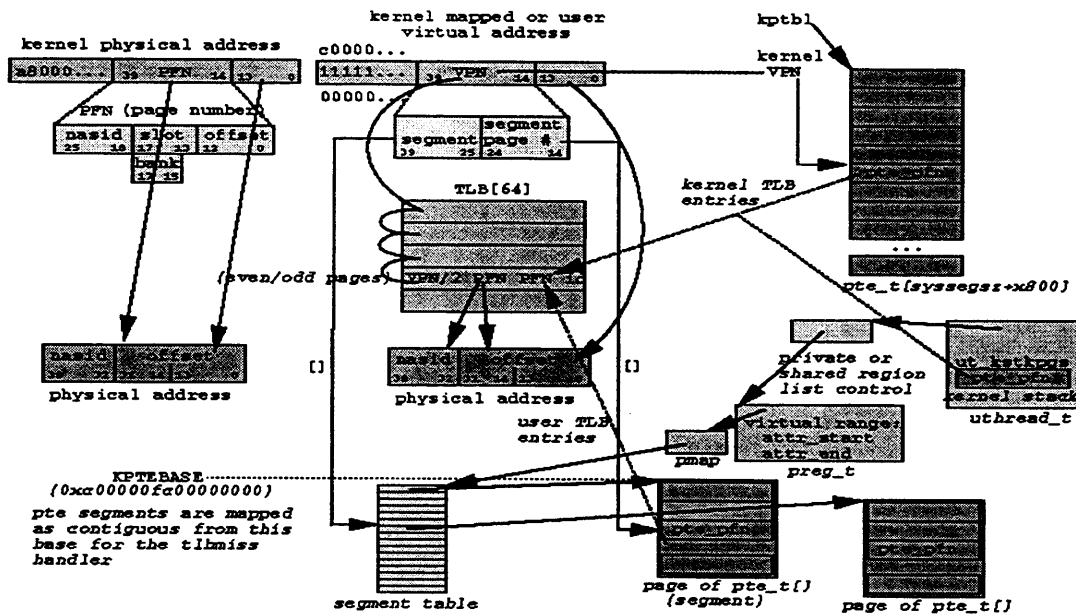
>> pfdat -n
VALID PFNS :

Node : 0 0 -- 8191 (8192)
Node : 0 32768 -- 40959 (8192)
Node : 0 65536 -- 73727 (8192)
Node : 0 98304 -- 106495 (8192)
Node : 0 131072 -- 135167 (4096)
Node : 0 163840 -- 167935 (4096)
Node : 0 196608 -- 200703 (4096)
Node : 0 229376 -- 233471 (4096)
Node : 1 262144 -- 270335 (8192)
Node : 1 294912 -- 303103 (8192)
Node : 1 327680 -- 335871 (8192)
...

```

The memory at the base of each node is reserved for (?)

Address Translation



kernel physical addresses

- begin with "a8000000..."

- are used by the kernel to address most dynamically allocated data
- are not translated through the TLB

kernel mapped addresses

- begin with "c0000000..."
- are used by the kernel to address kernel text (mapped to the copy on each node0) and data from the unix binary
- a TLB miss causes the kernel to go to the kernel page table (kptbl->) for a pte to load into the TLB

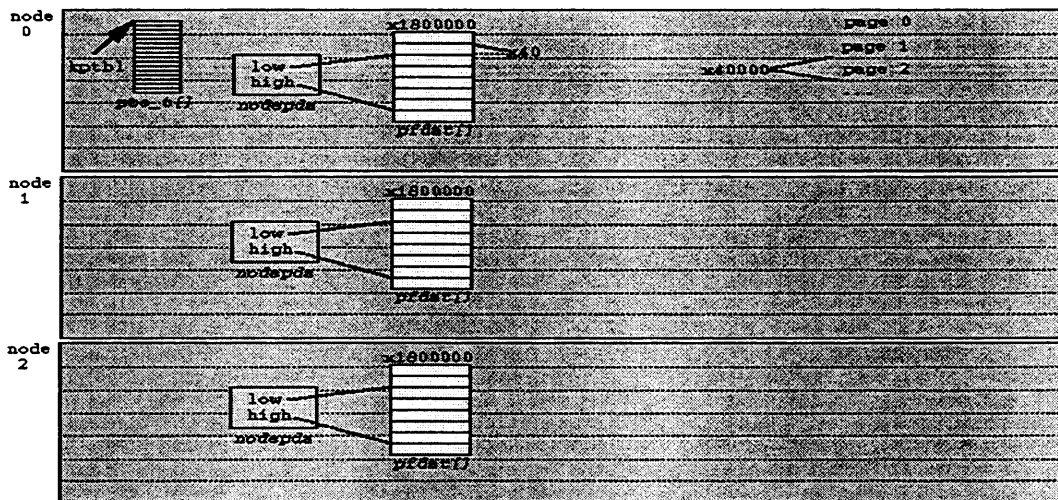
kernel stack addresses (mapped)

- begin with "ffffff..."
- are used by the kernel to address the kernel stack associated with the current thread
- the pte_t for the kernel stack is kept in the utthread_t
- no TLB miss should occur
- the virtual base of each kernel stack is ffffffff8000 (KSTACKPAGE)

user virtual addresses

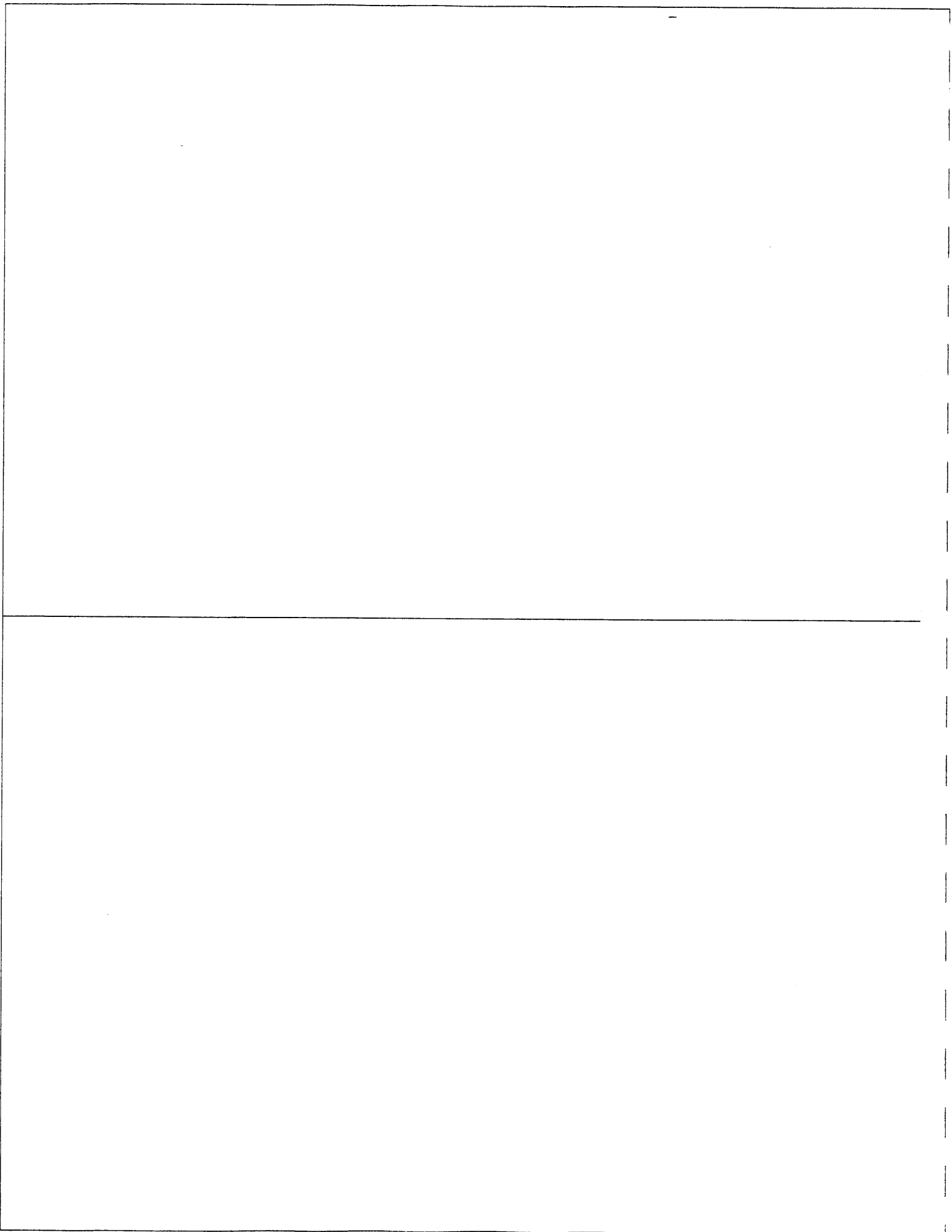
- begin with "00000000..."
- a user may only use a virtual address (no kernel addressing modes are permitted by the hardware)
- a TLB miss causes the kernel to go to the current thread's preg_t's to find a virtual region containing the missed virtual address
- the kernel goes to this pregion's table of pte's for a pte to load into the TLB
- the kernel may use a user virtual address -- the kernel executes with the same address space id (tbpid, or asid) as the current user thread, so address translation is the same as in user mode
 - the kernel is given user addresses via system calls
- structure shown above assumes PMAP_SEGMENT which is used for a 32-bit mode binary
- For a PMAP_TRILEVEL (64-bit mode) process the segment table points to other segment tables which point to pages of pte's. The diagram does not illustrate this case.

Table locations



The pfn_64's are local to the nodes they describe. They are indexed by the low 18 bits of the PFN (i.e. excluding the nasid). They correspond positionally to the pages on the node.

There is only 1 kernel page table. It is physically located on node 0.



Module 15: Disk I/O

Example IRIX read(2) Sequence

```
1 read(2)
  ----- user->kernel-----
2 sysstrap
3 syscall
  --sysent[]---ml->os-----
4 read
5  _read
  VOP_READ
  -----xfs_vnodeops[]-----os->xfs-----
6  xfs_read
7    xfs_read_file
8    chunkread
  VOP_STRATEGY
  -----xfs_vnodeops[]-----
9    xfs_strategy
  (bp->bp_target set to mount mp->m_ddev_targp, which has a pointer
  to the bdevsw entry)
10   xfs_strat_read
11   xfsbdstrat (mp, bp) (use bp->b_target->bdevsw)
12   bdrv [bdstrat macro]

  -----bdevsw[]-----xfs->xlv-----
13     xlvstrategy (minor is index to subvolume)
14     xlv_lower_strategy
15     griostrategy (major 0: use hwgraph)
16     bdrv [bdstrat macro]

  -----bdevsw[]-----xlv->disk driver---
17     dkscstrategy
18     dksccommand
19     dkscstart

  ----lun vertex->target vertex->controller vertex->sciccommand
```

15-1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

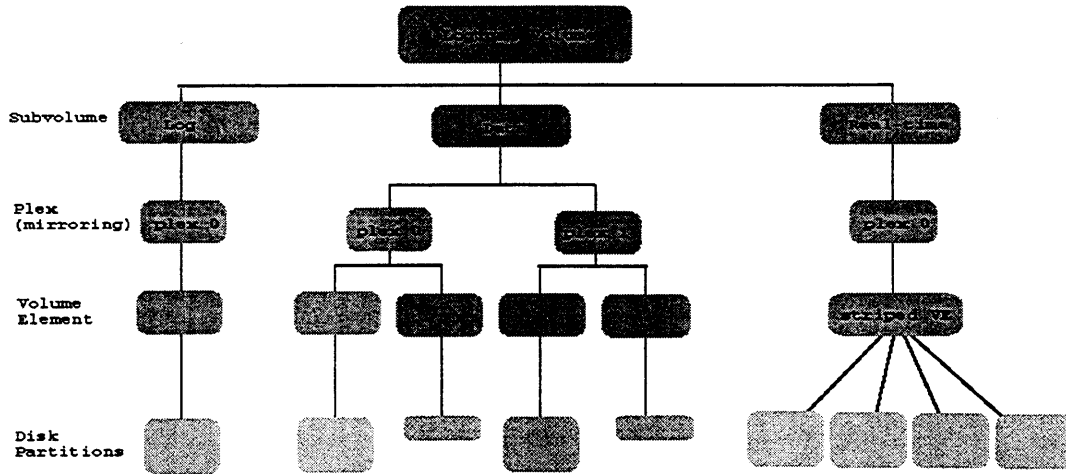
```
20     qlcommand
21     ql_entry
22     ql_start_scsi
23     ql_PCIOUTH moves to registers
```

15-1.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

XLV Structure



- see man xlv
- concatenated filesystem: make multiple VE's of 1 partition each
- striped filesystem: make 1 VE of multiple partitions
- the VE is the unit of recovery; i.e. on a disk error, the VE is taken offline
- xlv_make(1M) is used create logical volumes by writing their definitions to the labels of the disks that will constitute the logical volume objects
- xlv_assemble(1M) is run at boot time to collect the labels from all the disks, and pass them to the kernel

xlv_assemble(1M) also creates all the nodes in /dev/xlv; it assigns major 192 (XLV_MAJOR) and creates a minor number
 the kernel strings all the subvolumes together in one table (see below), with minor number being the index into that table

- example:

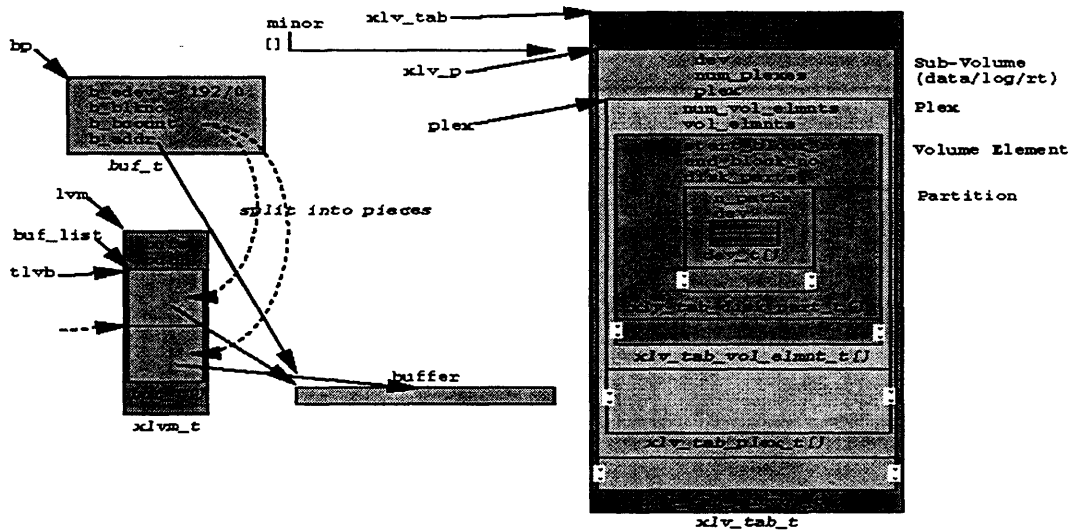
```
$ ls -la /dev/xlv
brw----- 1 root root 192, 8 Mar 4 10:43 kud_src
brw----- 1 root root 192, 4 Mar 4 10:43 opt
brw----- 1 root root 192, 9 Mar 4 10:43 people
brw----- 1 root root 192, 5 Mar 4 10:43 tmp
brw----- 1 root root 192, 7 Mar 4 10:43 tmp
brw----- 1 root root 192, 6 Mar 4 10:43 utmp
```

- to see your configuration:

```
$ xlv_mgr
xlv_mgr> show kernel
VOL opt flags=0x1, [complete] (node=NULL)
DATA flags=0x4(Block_IO) open_flag=0x3(FREAD|FWRITE) device=(192, 4)
PLEX 0 flags=0x0
VE 0 [active]
start=0, end=35554815, (stripe)grp_size=2, stripe_unit_size=64
/hw/module/7/slot/iol/baseio/pci/0/scsi_ctlr/0/target/4/lun/0/disk/partition/7/block (1777424 blks)
/hw/module/8/slot/iol/baseio/pci/0/scsi_ctlr/0/target/4/lun/0/disk/partition/7/block (1777424 blks)
.....
```

- also see xlv_shutdown, prtvtoc, fx, dvhtool

XLV Driver Layer



`xlvrstrategy(bp, 0) /* assume a single plex */`

- minor device is taken from `bp->b_edev`
- `xlvr_p = &xlvr_tab->subvolume[minor_dev];`
- if `num_plexes > 1`, split the I/O among the volume's plexes (i.e. mirrors)

15-3

22jul1998

TR-IKI rev 0.7b SGI Proprietary

- `xlvr_lower_strategy (bp, 0);`

`xlvr_lower_strategy(bp, plex_num)`

- `plex = xlvr_p->plex[plex_num];`
- calculate the number of buffers needed to do the transfers to the real disk partitions
- allocate an array of `buf_t` structures for the transfers ("lvm")
- break the `bp->buf` up into separate requests in `buf_list->buf's`
`b_edev` will be a partition device number (i.e. a hwgraph vertex handle)
- for each `buf (tlvb)` in list
`griostategy(tlvb)`

`griostategy(bp)`

- check for guaranteed rate I/O information on this (`b_edev`) disk; if none:
- `my_bdevsw = get_bdevsw(bp->b_edev); /* this macro assumes b_edev is a vertex if major is 0, else indexes into bdevsw[] */`
- `bdstrat(my_bdevsw, bp); /* this a macro that calls bdrv(my_devsw,DC_STRAT,bp) */`

`bdrv(bdevsw *my_bdevsw, int routine, ...)`

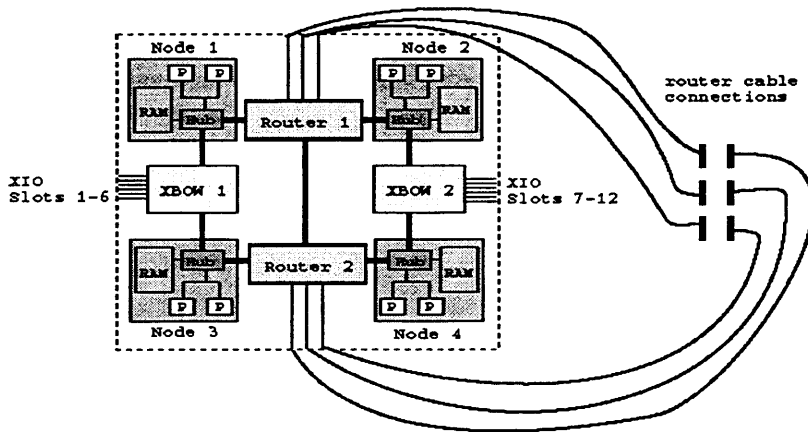
- case `DC_STRAT`: `func = (bdevfunc_t)my_bdevsw->d_strategy;`
- `(*func)(a1, a2, a3, a4, a5); /* calls disk driver dkscstrategy(bp) */`

15-3.a

22jul1998

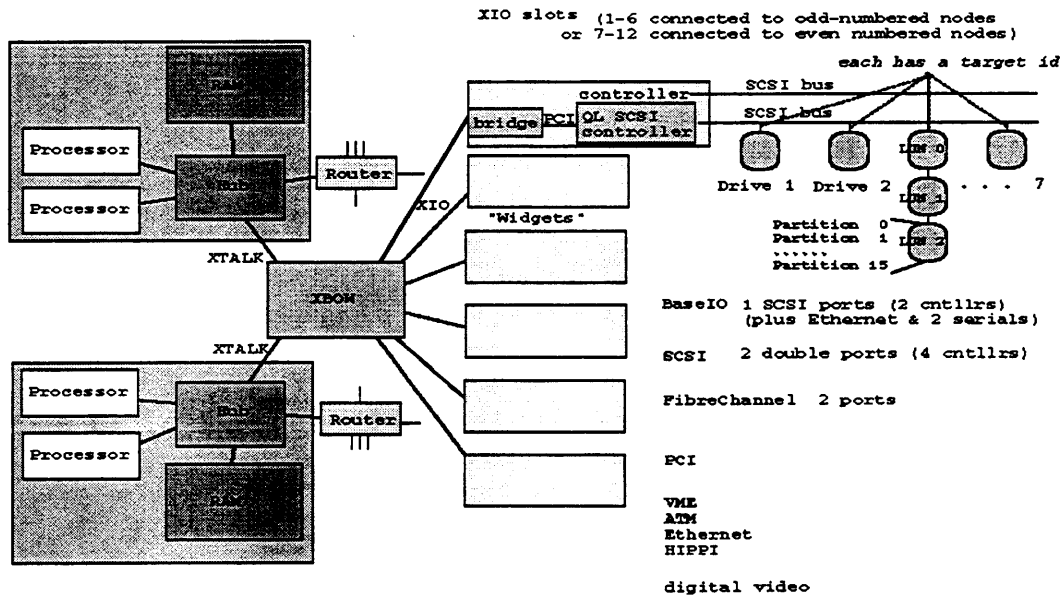
TR-IKI rev 0.7b SGI Proprietary

ORIGIN module overview



- The diagram above illustrates how the 12 I/O slots are connected to the 8 processors in an ORIGIN module

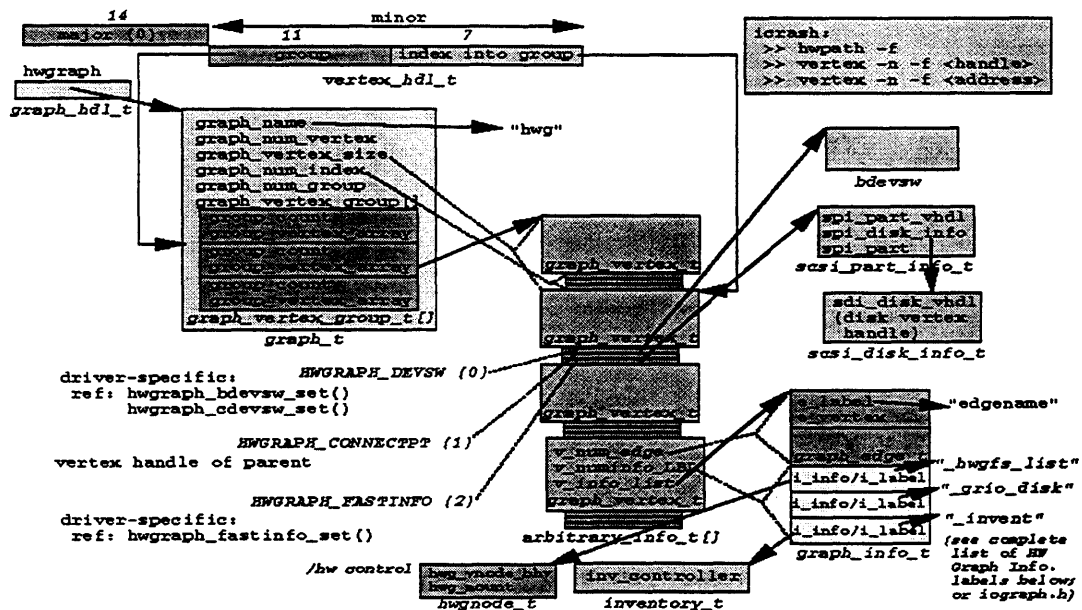
Disk Device Connections to be Pictured in the Hardware Graph



- half of a module is pictured above
- the OS probes the hardware at startup time and builds the hardware graph in memory
- the hwgfs file system is mounted on /hw

- each controller is assigned a unique number by ioconfig(1M) at boot time
 - before mounting filesystems (called by /etc/bcheckrc)
 - controller numbers stored in /etc/ioconfig.conf
 - compare hinv output and /etc/ioconfig.conf to /hw, /dev/dsk and /dev/rdisk
- disk device naming conventions: **dksCNTRLdDRIVElLUNsPARTITION**
 - example /dev/dsk/dks14d5s7 is a SCSI disk in controller 14, drive (or "target") 5, [only logical unit 0], partition 7
 - "QL" is for QLogic scsi controller
 - "jag" for scsi via Jaguar VME; "fd" for floppy disk
- partition conventions:
 - 0 - root
 - 1 - swap
 - 6 - usr
 - 7 - entire disk except header and log
 - 8 - volume header
 - 9 reserved (bad blocks)
 - 10 - entire disk
 - 11 - xfs log (external)
- disk labels:
 - controller parameters; e.g. device geometry
 - partition map
 - sgi info; e.g. serial number
 - boot info; e.g. root and swap partitions; name of file to boot
 - directory; e.g. sash

Hardware Graph format



- group 0 index 1 (i.e. handle 1) is the root of /hw
- built dynamically in kernel memory at boot time

- accessible externally as the /hw filesystem
- meaning of a hwgraph "minor device number":

```
$ cd /hw/disk
$ls -l a root swap
brw----- 1 root      0,3842 Mar  2 17:45 root
brw----- 1 root      0,3847 Mar  1 08:41 swap

$ dtb 3842
11110 0000010 /* 1E 2 i.e. group 0x1E vertex 2 */
$ dtb 3847
11110 0000111 /* 1E 7 */
```

- using icrash to display the hwgraph control structure
 - each vertex has a name
 - below the names: 1=>2 means vertex handle 1 points to handle 2

```
>> hwpath -f /* be patient - this takes a while! */
/hw/module/1/elsc/nvram
 1=>2=>3=>4=>5
/hw/module/1/elsc/tty
 1=>2=>3=>4=>343
/hw/module/1/slot/nl/node/memory/.master
 1=>2=>3=>27=>28=>29=>31=>29
/hw/module/1/slot/nl/node/cpu/a/.master
 1=>2=>3=>27=>28=>29=>32=>33=>29
/hw/module/1/slot/nl/node/cpu/b/.master
 1=>2=>3=>27=>28=>29=>32=>35=>29
/hw/module/1/slot/nl/node/hub/.master
 1=>2=>3=>27=>28=>29=>36=>29
...

>> dump hwgraph
c0000000013daba0: a800000000da8000 |.....

>> print *(graph_t *)a800000000da8000
struct graph_s {
  graph_name = 0xa800000000dac008 = "hwg"
  ...
  graph_num_group = 128
```

```
graph_num_index = 3
graph_num_vertex = 2002
...
graph_vertex_size = 48
...
graph_vertex_group = {
  [0] graph_vertex_group_t {
    ...
    group_count = 128
    group_vertex_array = 0xa800000000daa000
  }
}
```

- using icrash to display all the vertices in the hwgraph

```
>> vertex -n -f | head -n 30
HNDL      VERTEX  REFCNT  NUM_EDGE  NUM_INFO  INFO_LIST
-----
1 a80000000131e030 33      30        1 a800001000501400
```

```
CONNECT_POINT=0
INDEX[0]=0x0, INDEX[1]=0x0, INDEX[2]=0x0
```

EDGES:

EDGE	LABEL	NAME	VERTEX
0	a800000001360968	module	2
1	a8000000013613a0	nodenum	70
2	a800000001361490	cpunum	74
3	a800000001362f18	xplink	713
4	a800000001362f78	midi	714
5	a800000001363200	video	767
6	a8000002004a4098	.id	833
7	a8000002e004a4140	.invent	1596
8	a8000001004a4e60	machdep	2750
9	a8000001004a53a0	ttys	3023
10	a8000001004a5898	unknown	3684
11	a8000001004a43f8	disk	3844
12	a8000001004a5aa8	rdisk	3845
13	a8000001004a5748	.devhdl	3849

```

14 a8000000248d100 external_int 3851
15 a8000000248d130 sharena 3867
16 a8000000248d178 zero 3868
17 a8000000248d1c0 mem 3869
18 a8000000248d1d8 kmem 3870

```

- using icrash to display a disk vertex
 - 2305 is binary 10010 0000001
 - group id is 0x12
 - group index is 0x1

```

>> vertex -n -f 2305
HNDL VERTEX REFCNT NUM_EDGE NUM_INFO INFO_LIST
-----
2305 a800001a0050c030 6 3 2 a800001c264486c0

```

```

CONNECT_POINT=2679
INDEX[0]=0x0, INDEX[1]=0xa77, INDEX[2]=0xa8000018004b0330

```

EDGES:

EDGE	LABEL	NAME	VERTEX
0	a8000001004a42d8	volume	2690
1	a8000002004a4320	volume_header	2693
2	a8000003004a4128	partition	3678

INFOS:

INFO	LABEL	NAME	INFO_DESC	INFO
0	a800000001363080	_invent	20 a8000018004ac460	
1	a80000000248d298	_hwgfs_list	ffffffffffffffff a800001c0050a280	

```

=====
1 graph_vertex_s struct found

```

- meaning of icrash "vertex -l -n" headings:
 - HNDL: decimal handle of this vertex

15-6.c

22jul1998

TR-IKI rev 0.7b SGI Proprietary

- VERTEX: hex address of this vertex
- REFCNT: reference count
- NUM_EDGE: number of graph_edge_t's
- NUM_INFO: number of graph_info_t's
- INFO_LIST: hex address of the list of graph_info_t's and graph_info_t's
- CONNECT_POINT: parent handle (arbitrary_info_t[1] following the graph_vertex_t)
- INDEX[0]: hex; arbitrary_info_t[0] following the graph_vertex_t; can be a pointer to driver table
- INDEX[1]: hex; arbitrary_info_t[1] following the graph_vertex_t; same as CONNECT_POINT
- INDEX[2]: hex; arbitrary_info_t[2] following the graph_vertex_t; "fastinfo"
 - the arbitrary_info_t's:

```

/* Reserve room in every vertex for 3 pieces of fast access indexed information */
#define HWGRAPH_NUM_INDEX_INFO 3
#define HWGRAPH_DEVSW 0 /* (b,c)devsw for this device */
#define HWGRAPH_CONNECTPT 1 /* connect point (parent) */
#define HWGRAPH_FASTINFO 2 /* reserved for creator of vertex */

```

- for disk-related devices HWGRAPH_FASTINFO (index 2) points to:

```

scsi controller scsi_ctlr_info_t
scsi target scsi_targ_info_t
scsi lun scsi_lun_info_t
scsi unit scsi_unit_info_t
scsi device scsi_dev_info_t
scsi disk scsi_disk_info_t
scsi partition scsi_part_info_t

```

- EDGES: graph_edge_t's
- EDGE: index into the graph_edge_t's
- LABEL: hex address of the graph_edge_t
- NAME: e_label->character string
- VERTEX: decimal vertex handle of the subordinate (child) vertex
- INFOS: graph_info_t's
- INFO: index into graph_info_t's
- LABEL: hex address of the graph_info_t
- NAME: i_label->character string (see HW Graph Information Labels below for full list)
- INFO_DESC:
 - -1: INFO_DESC_PRIVATE - info is not exported as an attribute on a /hwgfs file

15-6.d

22jul1998

TR-IKI rev 0.7b SGI Proprietary

- 0: INFO_DESC_EXPORT - info is in the arbitrary_info_t itself
- NN: size of the structure that i_info points to
- INFO: i_info; the arbitrary_info_t within the graph_info_t; this is either information or a pointer to information

- jump-off for further study: example code sequence that sets a bdevsw pointer in to the hwgraph

```

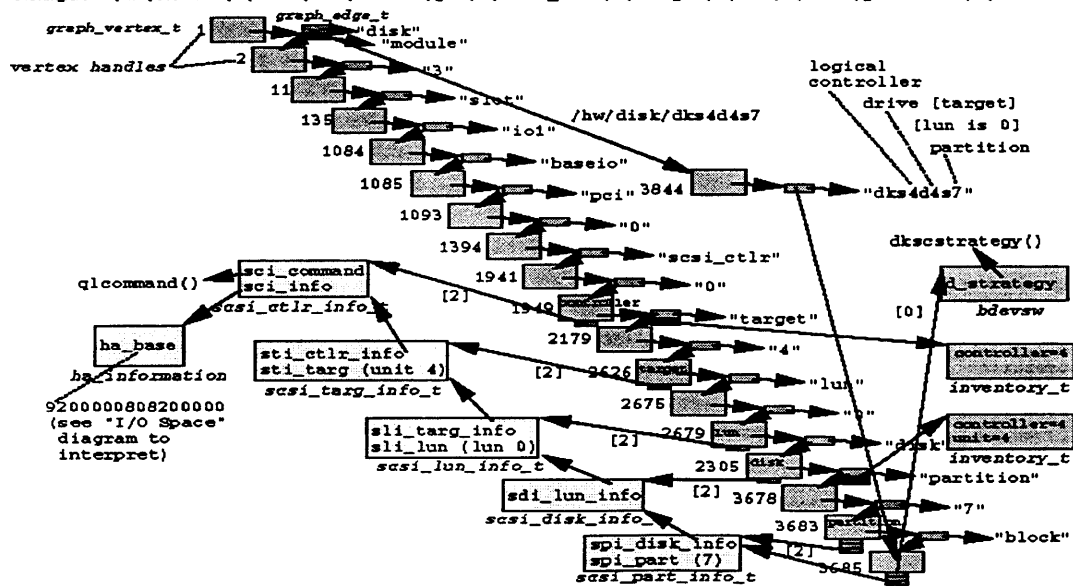
main
initialize_io
edt_init
edt[]
wd93edtinit
wd93hinw
wd93inq
scsi_device_update
scsi_disk_update
scsi_part_vertex_add
hwgraph_device_add
hwgraph_block_device_add
hwgraph_bdevsw_set

```

Hwgraph Example

(from vmcore.320.comp)

example: /hw/module/3/slot/io1/baseio/pci/0/scsi_ctlr/0/target/4/lun/0/disk/partition/7/block



● associating partition name and physical location with icrash

```
>> hwpath -f dks4d4s7 /* to find the partition's vertex */
/hw/disk/dks4d4s7
  1=>3844=>3685 /hw/rdisk/dks4d4s7
  1=>3845=>3692
```

○ walk backward from 3685 to its controller vertex

```
>> vertex -f 3685 | grep CONN
CONNECT_POINT=3683

>> vertex -f 3683 | grep CONN
CONNECT_POINT=3678

>> vertex -f 3678 | grep CONN
CONNECT_POINT=2305

>> vertex -f 2305 | grep CONN
CONNECT_POINT=2679

>> vertex -f 2679 | grep CONN
CONNECT_POINT=2675

>> CONNECT_POINT=2679
CONNECT_POINT=2679: unknown command

>> vertex -f 2675 | grep CONN
CONNECT_POINT=2626

>> vertex -f 2626 | grep CONN
CONNECT_POINT=2179

>> vertex -f 2179 | grep CONN
CONNECT_POINT=1949

>> vertex -f 1949 | grep CONN
CONNECT_POINT=1941

>> vertex -f 1941 | grep CONN
CONNECT_POINT=1394
```

15-7.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
>> vertex -f -n 1394
HNDL          VERTEX  REFCNT  NUM_EDGE  NUM_INFO      INFO_LIST
=====
1394 a800001500509560      17        10         4 a800001c65477800
CONNECT_POINT=1093
INDEX[0]=0x0, INDEX[1]=0x445, INDEX[2]=0xa8000008004f4400

EDGES:

EDGE          LABEL  NAME          VERTEX
-----
  0 a8000000013613e8  .master      1093
  1 a8000005004a41e8  usrpqi      1407
  2 a8000001004a40e0  scsi_ctlr   1941
  3 a8000005004a4260  base        5281
  4 a8000002004a4050  io          5282
  5 a80000000248d1c0  mem         5283
  6 a8000005004a4218  config     5284
  7 a8000005004a4638  dma         5285
  8 a8000005004a46e0  intr       5286
  9 a800000041329940  rom         5287

INFOS:

INFO          LABEL  NAME          INFO_DESC      INFO
-----
  0 a8000005004a41b8  _pciio      ffffffff a8000008004f4400
  1 a800000001363080  _invent      20 a8000008004ac100
  2 a8000002004a4140  _device_desc ffffffff a8000008004ac4c0
  3 a80000000248d298  _hwgfs_list ffffffff a800001c24e83c00

=====
1 graph_vertex_s struct found

>> vertex -n 1941
HNDL          VERTEX  REFCNT  NUM_EDGE  NUM_INFO      INFO_LIST
=====
1941 a80000150050e3f0      4         1         1 a800001c254873f0
```

```
EDGES:

EDGE          LABEL  NAME          VERTEX
-----
```

15-7.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

0 a80000001360980 0 1949
INFOS:
INFO LABEL_NAME INFO_DESC INFO
-----
0 a8000000248d298 _hwgfs_list ffffffff a80001c24e83480
=====
1 graph_vertex_s struct found

o Notice above that 1941 is a scsi_ctrl edge "0" leading to the controller vertex #1949:
o display the controller number for vertex 1949:

>> vertex -f -n 1949
HNDL VERTEX REFCNT NUM_EDGE NUM_INFO INFO_LIST
-----
1949 a80000150050e570 7 2 2 a800001c24e829a0

CONNECT_POINT=1941
INDEX[0]=0x0, INDEX[1]=0x795, INDEX[2]=0xa800000100504300

EDGES:
EDGE LABEL_NAME VERTEX
-----
0 a8000001004a4110 bus 1955
1 a8000001004a4128 target 2179

INFOS:
INFO LABEL_NAME INFO_DESC INFO
-----
0 a80000001363080 _invent 20 a8000018004ac1c0
1 a8000000248d298 _hwgfs_list ffffffff a80001c24e82a60
=====
1 graph_vertex_s struct found

/* the _invent label points to an inventory_s structure (of x20 bytes)*/

```

```

>> print *(inventory_t *)0xa8000018004ac1c0
struct inventory_s {
    inv_next = (nil)
    inv_class = 2 /* INV_DISK 2 */
    inv_type = 1 /* INV_SCSICONTROL 1 */
    inv_controller = 4 /* matches /hw/disk/dks4d4s7 and /dev/dsk/dks4d4s7 */
    inv_unit = 0
    inv_state = 9
}

```

o the controller logical number is assigned in conjunction with ioconfig(1M) (see ioctl (fd, DS_MKALIAS))

● Alternate method: dump all paths to a file:

```
>> hwpath -f | cat > hwpaths
```

o Search the file for the name and vertex number:

```
/hw/disk/dks4d4s7
1=>3844=>3685
```

o walk forward in this list to the scsi_ctrl:

```
/hw/module/3/slot/io1/baseio/pci/0/scsi_ctrl/0/target/4/lun/0/disk/partition/7/block
1=>2=>11=>135=>1084=>1085=>1093=>1394=>1941=>1949=>2179=>2626=>2675=>2679=>2305=>3678=>3683=>3685
```

o then display that vertex as above (>> vertex -f -n 1949)

HWGraph Information Labels

see iograph.h:

Symbolic name	ASCII label:	Contains:
INFO_LBL_ASYNC_ATTACH	"_async_attach"	async_attach_t
INFO_LBL_CNODEID	"_cnodeid"	cnodeid_t (compact node id)
INFO_LBL_CONTROLLER_NAME	"_controller_name"	&inventory_t
INFO_LBL_CPUID	"_cpuid"	cpuid_t
INFO_LBL_CPU_INFO	"_cpu"	&cpuinfo_t
INFO_LBL_DETAIL_INVENT	"_detail_invent"	&invent_cpuinfo_t &invent_miscinfo_t &invent_meminfo_t &invent_routerinfo_t
INFO_LBL_DEVICE_DESC	"_device_desc"	&device_desc_t
INFO_LBL_DIAGVAL	"_diag_reason"	&diag_inv_t
INFO_LBL_DKIOTIME	"_dkiotime"	&dksc_local_info_t
INFO_LBL_DRIVER	"_driver"	&device_driver_t not used?
INFO_LBL_ELSC	"_elsc"	not used?
INFO_LBL_GIOIO	"_gioio"	not used?
INFO_LBL_GFUNCS	"_gioio_ops"	&gioio_provider_t
INFO_LBL_GRIO_DSK	"_grio_disk"	&grio_disk_info_t
INFO_LBL_HUB_INFO	"_hubinfo"	&hubstat_t
INFO_LBL_HWGFSLIST	"_hwgfs_list"	&hwgnode_t
INFO_LBL_TRAVERSE	"_hwg_traverse"	&traverse_fn_t
INFO_LBL_INVENT	"_invent"	&inventory_t
INFO_LBL_MLRESET	"_mlreset"	not used?
INFO_LBL_MODULE_INFO	"_module"	not used?
INFO_LBL_MONDATA	"_mon"	&router_info_t &hubstat_t
INFO_LBL_MDPERF_DATA	"_mdperf"	&md_perf_monitor_t
INFO_LBL_NIC	"_nic"	char *
INFO_LBL_NODE_INFO	"_node"	&hubinfo_t
INFO_LBL_PCIBR_HINTS	"_pcibr_hints"	pcibr_hints_t
INFO_LBL_PCIIO	"_pciio"	pciio_info_t
INFO_LBL_PFUNCS	"_pciio_ops"	&pciio_provider_t

15-8

22jul1998

TR-IKI rev 0.7b SGI Proprietary

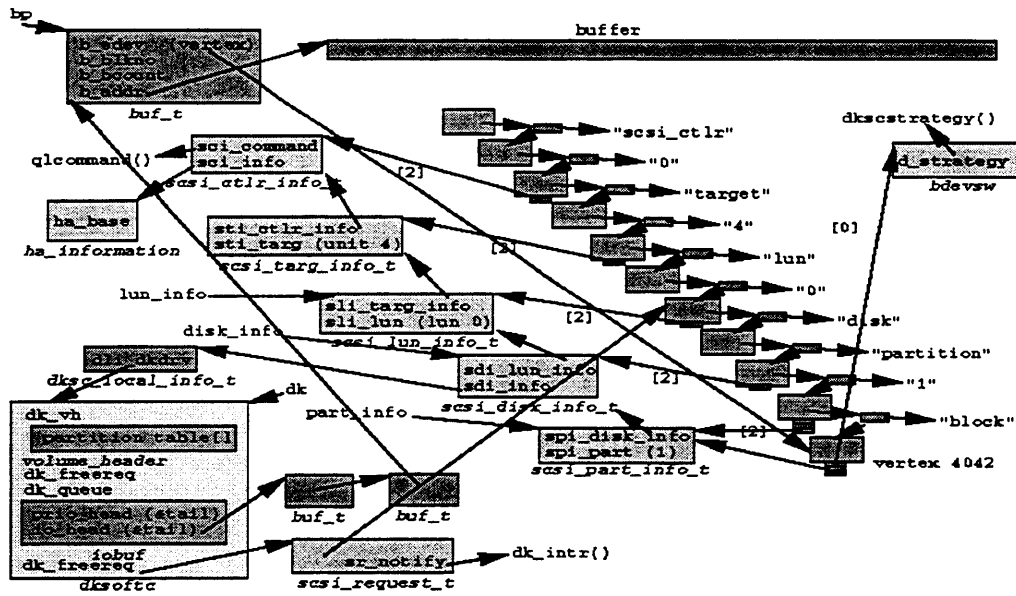
INFO_LBL_PERMISSIONS	"_permissions"	hwgfs_perm_t
INFO_LBL_ROUTER_INFO	"_router"	&router_info_t
INFO_LBL_SUBDEVS	"_subdevs"	&ulong_t
INFO_LBL_VME_FUNCS	"_vmeio_ops"	&vmeio_provider_t
INFO_LBL_XSWITCH	"_xswitch"	not used?
INFO_LBL_XSWITCH_ID	"_xswitch_id"	arbitrary_info_t xbow_num
INFO_LBL_XSWITCH_VOL	"_xswitch_volunteer"	&xswitch_vol_t
INFO_LBL_XFUNCS	"_xtalk_ops"	not used?
INFO_LBL_XWIDGET	"_xwidget"	xwidget_info_t

15-8.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Disk Driver Layer



dkscstrategy(bp)

- bp->b_edev is the disk partition's vertex in the hwgraph

(this is stored in the /hw/disk/dks.... node:

```
$ ls -la /hw/disk/dks0d4s1brw----- 1 root root 0,4042 Apr 27 11:23 /hw/disk/dks0d4s1
```

```
$ ls -la 'find /hw/module -name block -depth' | grep 4042
```

```
brw----- 1 root root 0,4042 Apr 27 11:26 /hw/module/6/slot/io1/baseio/pci/0/scsi_ctr/0/target/4/lun/0/disk/partition/1/block
```

[also see /etc/ioconfig.conf for path and controller number])

- set "part_info" to partition vertex's information
- set "disk_info" to disk vertex's information
- set "dk" to disk's queue control structure
- calculate disk-relative block number (into b_sort)
- buf may, or may not, be sorted into normal or priority queue
- dksccommand(dk, &sc); /* sc a pointer to lock dk->dk_lock */

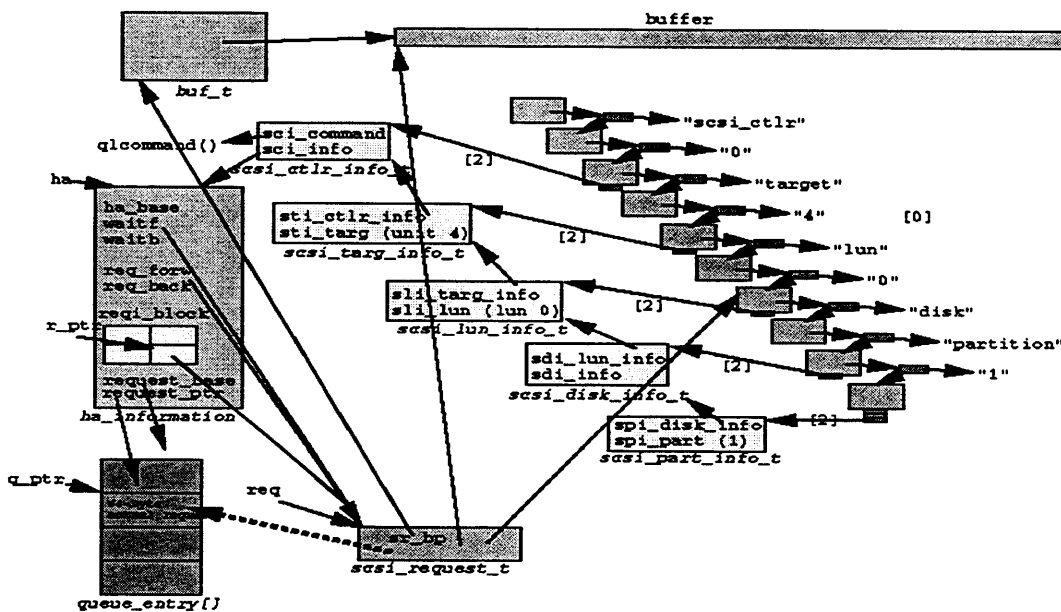
dksccommand(struct dksoftc *dk, int *sp)

- set bp to top request on the dk-> retry, priority and normal queues
- set sp to dk_freereq (a scsi request structure)
- dkscstart(dk, bp, sp);

dkscstart(struct dksoftc *dk, buf_t *bp, scsi_request_t *sp)

- set lun_info (follow scsi request to lun vertex)
- copy the scsi read or command from dk->structure to scsi request
- set addresses and length in to scsi request
- sr_notify set to dk_intr() for interrupt handler
- set interrupt handler (sr_notify) to dk_intr()
- find the controller vertex (follow lun to target to controller vertex) and call scsi driver (scsi_command)(sp) qlcommand(sp) /***** assuming a QLogic scsi controller *****/

SCSI Driver Layer



qlcommand(struct scsi_request *req)

- ql_entry(req);

ql_entry(scsi_request_t* request)

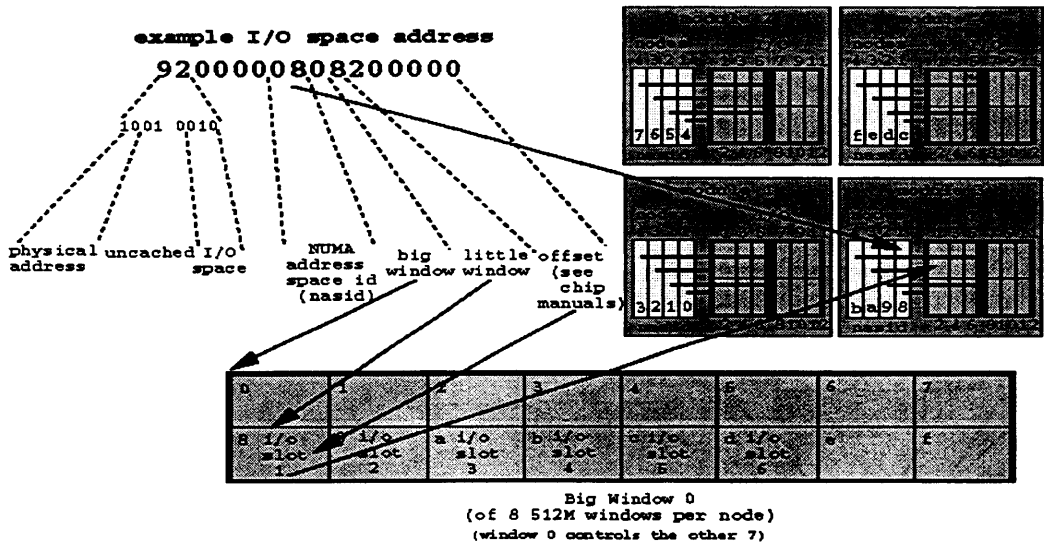
- ha = ha_information (hardware address)
(follow request's pointer to lun vertex, to target vertex, to controller vertex)
- initialize the request return status fields
- place the request on the ha->waitf/waitb wait queue
- ql_start_scsi(ha)

ql_start_scsi(pHA_INFO ha)

(preliminary description - key points may be missing)

- isp = ha->ha_base /* base address for communicating with this controller */
(see I/O Address Space, below)
- move any request from the ha->wait queue to the ha->req_forw/back request queue
- work on first request in ha->req_forw request queue
 - check controller's queue space
 - remove request from ha->req_forw queue, link to it from the ha->reqi_block[][]
 - build a request at q_ptr->queue_entry from request->scsi_request_t
 - adjust request for scatter/gather
 - inform controller of request: QL_PCI_OUTH(&isp->mailbox4, ha->request_in);
[*((volatile short*)&isp->mailbox4) = ha->request_in]
- move any other ha->wait request to ha->req_forw and repeat above

I/O Address Space

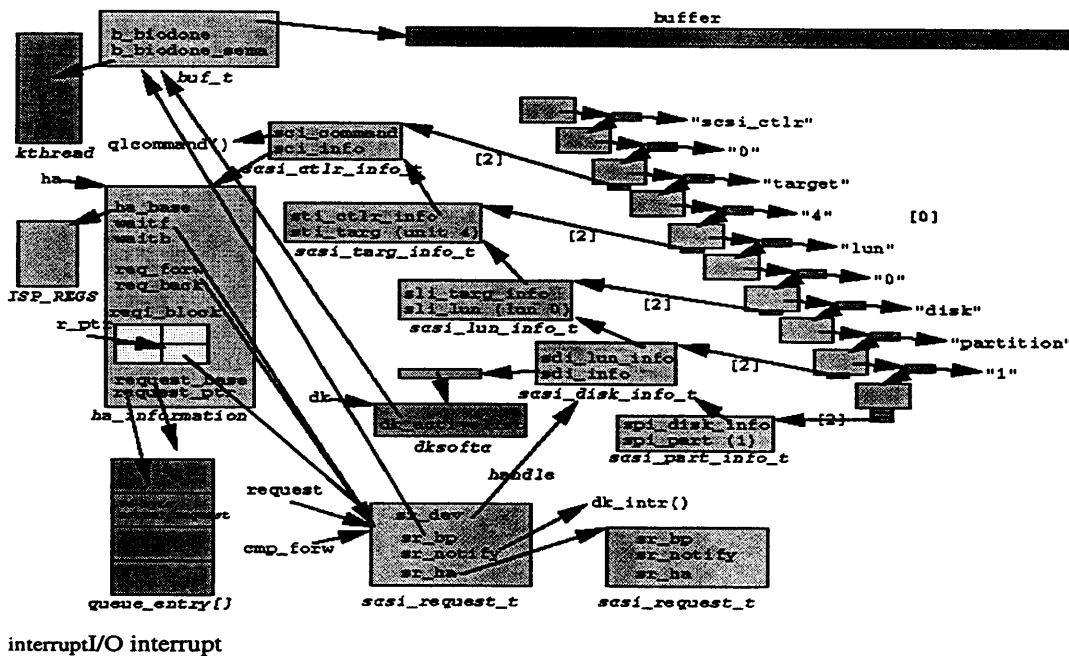


- Each cpu can address an I/O device at an "I/O" space address, as diagrammed above.
- nasid's: use "hinv -mv | grep Slice" to determine nasid assignments
- even numbered nodes are connected to high numbered I/O slots
- odd numbered nodes are connected to low numbered I/O slots
- see http://www-ssd.engr/doc/chips/hub2_prog_man/hub2.regbook.doc.html for I/O space addressing details

- see <http://www-ssd.egr.doc/> for the general table of contents
- see the IRIX Device Driver Programmer's Guide (6.4)
- see the IRIX Device Driver Programmer's Guide (6.5)
- Each xtalk device is accessible in the hwgraph via its I/O slot number, and via its controlling hub. For example:

```
$ ls -l /hw/module/1/slot/n1/node/xtalk/8
lrw----- 1 root root 29 Mar 6 11:11 /hw/module/1/slot/n1/node/xtalk/8 -> /hw/module/1/slot/io1/baseio
```

Interrupt Processing



interrupt/I/O interrupt

- CPU enters at E_VEC 0x80000180

exception

- code was copied here by _hook_exceptions
- is_intr processes an "interrupt"

intrfast

intrnorm

ecommon

- pda's p_causevec[CAUSE exception code]
- this table was copied to pda from causevec[]

VEC_int(ep)

intr(ep, ...

- if (cause & SRB_DEVO) (devices bit)
(cause_intr_tbl[SRB_DEVO_IDX])(ep) (calls intpend0 for device interrupt)

intpend0(ep)

- while Hub PI_INT_PEND0
 - level = leading bit
 - clear the bit
 - pda: p_intmasks.dispatch0->vectors[level]

TR-IKI rev 0.7b SGI Proprietary

22jul1998

15-12.a

```

iv_func |----> qlintr()
iv_args |----> ha_information
iv_sync |-----
-----

```

- if THD_OK (i.e. threaded, which is normal)
vsema (&iv_sync) (wakeup intpend0_intrd)

intpend0_intrd(intr_vector_t * ivp) (interrupt thread)

- (iv_func)(iv_arg) (call to qlintr(&ha_information))
- ipsema (&iv_sync)

qlintr(pHA_INFO ha)

- read ha_base->bus_isr to test for interrupt
(ISP_REGS: register definitions)

ql_service_interrupt(ha)

- process "mailbox" request completions first
- pull all responses (ha->response_out to ha_response_in) to a linked list
- inform the adapter that they are received
- completion processing:
 - ql_notify_responses(ha, cmp_forw)
- start any more requests on ha->waif or ha->reqforw (ql_start_scsi())

ql_notify_responses(ha, forwp)

- for each request on sr_ha list:
 - (*request->sr_notify)(request)

TR-IKI rev 0.7b SGI Proprietary

22jul1998

15-12.b

dk_intr(sp)

- bp = sp->sr_bp
- remove buf from dk->dk_active_list

biodone(bp)

- check for guaranteed rate I/O
- if bp_biodone present, call (*bp_biodone)()
- set B_DONE
- vsema(&b_iodone_sema)

vsema(sema_t *sp)

- increment semaphores lock count

semawake(sp)

- find the thread (kt = semadq(sp))
- make the thread runnable (thread_unblock(kt, ...))

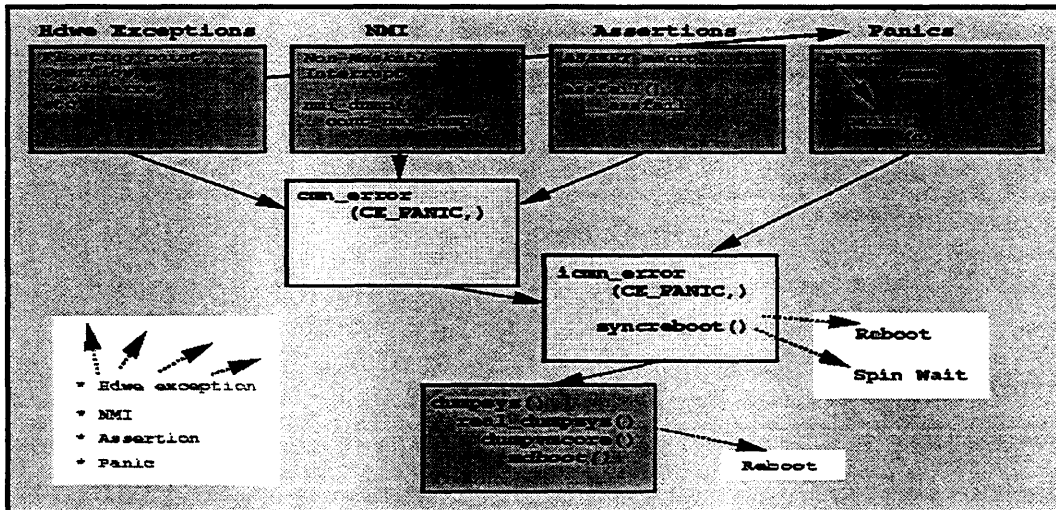
Module 16: IRIX Dumps - 6.5

IRIX Dumps - 6.5

This module describes:

- Situations in IRIX that cause system dumps
- Dump creation overview
- Dump contents
- Configuring dump levels

Dump Scenario Diagram



Dump Scenarios

Four causes

- **Hardware exceptions**

Interrupts caused by hardware conditions

For more serious machine errors, low level interrupt handlers call `panic`, or `cmn_err()` directly with `CE_PANIC` argument

Root cause may be:

- Hardware malfunction: eg. ECC (Error Correction Code) error; double bit error
- Software; BadVAddr (Bad Virtual Address); kernel address pointer error

Resulting panic message provides abbreviated reason for interrupt

- **NMI (Non-Maskable Interrupt)**

NMI interrupt triggered by operator pressing switch, or (remote) console request

Usually result of degraded or hung system

Way to "force" a dump for diagnostic purposes

Resulting panic message indicates "nmi" as cause

- **Assertions**

ASSERT macros scattered throughout kernel code test system "sanity"

Commonly conditionally compiled (`#ifdef`); used for testing / debugging

Resulting panic message includes failed test (source code expression)

- **Panics**

PANIC (assembler) macro calls `panic()` function

Panics typically indicate a logic error within the kernel; a very few may be caused by hardware

Resulting panic message provides abbreviated reason for interrupt

Common code

- All panic conditions call common error handling routine `icmn_err()` with the `CE_PANIC` argument
- Function `icmn_err()` invokes the system dump sequence for the `CE_PANIC` condition
- Function sequence `syncboot -> dumphsys() -> real_dumphsys -> dumpvmcore:`

 - Flushes memory cache
 - Flushes buffer cache areas to preserve file integrity
 - Saves CPU context data in memory
 - Write selected portion of virtual memory to the dump device
 - Call `mdboot()` to attempt to restart the system

Panic within panic

Hardware exceptions, NMI interrupts, ASSERT macro calls, PANIC macro calls, and panic calls that occur during panic processing result in corresponding handling routines to be called. Function `icmn_err()` detects this nested panic condition and:

- Dump processing is terminated; then ->
- A system reboot is attempted; if this fails ->
- Persistent re-entries of `icmn_err()` results in hanging all CPUs in a hard loop

Processing Activities

Hardware Exception

Processing

1. Low level kernel interrupt handler routines "trap" hardware errors and call functions such as `vec_int()` and `intr()`.
2. Function `intr()` checks the cause for the interrupt:
 - Errors affecting a user application usually only result in killing the application. The error is logged; a user core file may result.
 - Errors that are deemed "correctable" are corrected. The error is logged; the system continues.
 - Errors that compromise the integrity of the system are logged; then the system panics.
3. Depending on the situation, hardware routines may cause a panic by calling `icmn_err()` with the `CE_PANIC` argument, by executing `ASSERT` or `PANIC` macros, or by calling the `panic()` function.

In each case a panic message suggesting the reason for the panic is passed in the call.

4. In any of the above cases, eventually function `icmn_err()` is called process the panic (see below).

Panic Example

```
<0>PANIC: CPU 28: Fatal Craylink error.  
/hw/module/4/slot/n4/node/cpu/a: 1: RR_ERROR_STATUS = 0x9829000100041bf0  
NOTICE - cpu 30 didn't dump TLB, may be hung
```


Stack Trace Example

```
>> ctrace 28
=====
STACK TRACE FOR CPU 28

1 sync reboot[./os/printf.c: 1195, 0xc000000001bb030]
2 icmn_err[./os/printf.c: 342, 0xc000000001b99cc]
3 cmm_err[./os/printf.c: 116, 0xc000000001b91a0]
4 hubni_error_handler[./ml/SN0/huberror.c: 551, 0xc0000000002c020]
5 hubni_eint_handler[./ml/SN0/huberror.c: 561, 0xc0000000002c054]
6 handle_intpend1[./ml/SN0/intr.c: 990, 0xc00000000023518]
7 intr[./ml/SN0/intr.c: 1232, 0xc000000000237bc]
8 vec_int[./ml/LOCORE/vec_int.s: 80, 0xc00000000019418]
  r0/zero:0000000000000000  r1/at:0000000013eb8f64  r2/v0:0000000010efb580
  r3/v1:0000000010f2f190  r4/a0:0000000010efb610  r5/a1:00000000009ea88
  r6/a2:0000000010f35a20  r7/a3:0000000010eb9600  r8/a4:0000000010e978e0
  r9/a5:0000000010e85500  r10/a6:0000000014710ba4  r11/a7:00000000009dbcc
  r12/t0:000000001465ebf4  r13/t1:000000001166619c  r14/t2:00000000152305f0
  r15/t3:0000000014448c30  r16/s0:0000000000000049  r17/s1:000000001212efe0
  r18/s2:0000000014396c80  r19/s3:0000000010efb510  r20/s4:000000001360a678
  r21/s5:00000000136bc628  r22/s6:000000001376e5d8  r23/s7:0000000010eb0850
  r24/t8:0000000010dea8e0  r25/t9:000000000056cac  r26/k0:0000000000000000
  r27/k1:0000000000000020  r28/gp:00000000404004324  r29/sp:000000ffffa6ffbb0
  r30/s8:0000000010f2f290  r31/ra:0000000010ebbb20  EPC:00000000102bec34
  CAUSE=800, SR=fffffffa400ffb3, BADVADDR=12bf3668
```

NMI (Non-Maskable Interrupt)

Processing

1. NMI is a hardware interrupt, and thus is trapped by low level kernel code the same as other hardware interrupts. However the kernel takes a unique logic path for NMI in an attempt to save as much CPU context as possible for the dump.
2. Assembler routine `nmi_dump` sets up a dump stack area, and jumps to `cont_nmi_dump`.
3. C function `cont_nmi_dump`:
 - Places the system in panic (dump processing) state.
 - Indicates a NMI occurred: `nmied=1`.
 - Saves CPU registers in low memory ; `IP27_NMI_EFRAME_OFFSET=0x11800`
 - Collects router error information; print and log any problems found.
 - Calls `cmm_err(CE_PANIC, "User requested vmcore dump (NMI), cpu ___ handling"`.
4. Function `icmn_err()` prints panic message and processes dump (see below).

Panic Example

No usable example, yet.

Stack Trace Example

No usable example, yet

Assertions

Processing

There are many different ASSERT type macros coded in IRIX, however they all work around the same principle.

Consider two definition samples from `irix/kern/sys/debug.h`:

```
#define ASSERT(EX) ((!doass||)(EX)?((void)0):assfail(#EX, __FILE__, __LINE__))
#define ASSERT_ALWAYS(EX) ((EX)?((void)0):assfail(#EX, __FILE__, __LINE__))
```

Usage examples from `irix/kern/sgi/fs_bio.c`:

```
ASSERT(!bp->b_vp);
ASSERT(bp->b_blkno == blkno && bp->b_bcount == BBTOB(len));
```

Also from `irix/kern/os/fdt.c`:

```
ASSERT_ALWAYS(fp->vf_count > 0);
```

- Macro **ASSERT** is conditionally compiled into kernels with **#define DEBUG** set, hence it is used in development debugging.

Variable **doassert** is set to "1" (true) by default.

- Macro **ASSERT_ALWAYS** is unconditionally compiled into every kernel.
- In any case, assertions perform the following:
 1. Evaluate the expression "**ex**" for "true" or "false" (non-zero or zero).
 2. If the expression is true return void (0); effectively a NO-OP.
 3. If false call function **assertfail** with arguments the ASCII expression, the source code file name and line number.
 4. Assembler routine **assertfail** saves the calling CPU's registers in an exception frame called **_assertregs** and calls c function **_assertfail()**.
 5. Function **_assertfail**:
 1. Sets spin lock 7; forces other CPU to spin on this lock.
 2. Prints assertion message:

```
"assertion failed cpu CPUN: , file: FILE, line: LINE".
```

3. Prints values in **_assertregs**.
4. Sets system in panic mode.
5. Calls **cmd_err(CE_PANIC, "assertion failure!")**
6. **cmd_err()** prints the panic message and processes the dump (see below)

Panic Example

```
<6>assertion failed cpu 4: fp->vf_count > 0,
  file: /os/ldt.c, line: 100
<6>at/v0/v1:      0x0 0xffffffffffffbe90 0xffffffffffffc000
<6>a0-a3:         0x0 0x0 0x0 0x0
<6>a4-a7 (t0-t3): 0x0 0xc0000000001cdf40 0x0 0x1
<6>t0-t3 (t4-t7): 0xc00000000208ab20 0x0 0x16 0x0
<6>s0-s3:         0xa800000103897000 0xc000000001419bb8 0xa0 0xa80000010261b000
<6>s4-s7:         0x0 0x0 0x0 0x0
<6>t8/t9/k0/k1:   0x0 0xfb00f14 0x0 0x1
<6>gp/sp/fp/ra:   0xc000000001448d50 0xffffffffffffbe40 0x0 0xc000000001e1788
<6>mdlo/mdhi/sr:  0x0 0x2 0xffa1
<6>cause/epc/badv: 0x20 0xfb00f18 0xc000000002140020
<0>PANIC: CPU 4: assertion failure!
```

Stack Trace Example

```
>> ctrace 4
=====
STACK TRACE FOR CPU 4

1 dumpsys[./os/vmdump.c: 168, 0xc000000002399e4]
2 syncreboot[./os/printf.c: 1525, 0xc000000001d3ec4]
3 icmn_err[./os/printf.c: 554, 0xc000000001d28c4]
4 cmn_err[./os/printf.c: 149, 0xc000000001d1e18]
5 _assfail[./os/printf.c: 1825, 0xc000000001d44e0]
6 getf[./os/fdt.c: 100, 0xc0vncalls.c: 1673, 0xc000000001cdf50]
8 syscall[./os/trap.c: 2737, 0xc0000000018cba4]
9 sysstrap[./ml/LOCORE/sysstrap.s: 314, 0xc00000000037c48]
  r0/zero:0000000000000000    r1/at:000000007fffffff    r2/v0:0000000000000488
  r3/v1:0000000000000001    r4/a0:0000000000000002    r5/a1:0000000000000004
  r6/a2:000000007fff2050    r7/a3:0000000000000000    r8/a4:0000000000000000
  r9/a5:0000000000000000    r10/a6:0000000000000000   r11/a7:0000000000000000
  r12/t0:c000000002089d70   r13/t1:0000000000000000   r14/t2:00000000000068a1
  r15/t3:0000000000000001   r16/s0:0000000000000003   r17/s1:000000007fff2f24
  r18/s2:000000007fff2f34   r19/s3:000000007fff2fc4   r20/s4:0000000000000000
  r21/s5:0000000000000000   r22/s6:0000000000000000   r23/s7:0000000000000000
  r24/t8:0000000000000000   r25/t9:00000000fac06b8    r26/k0:0000000000000000
  r27/k1:00006205000d812a   r28/gp:00000000fb58134    r29/sp:000000007fff1f10
  r30/s8:0000000000000000   r31/ra:00000000fb03080    EPC:00000000fb00f1c
  CAUSE=1000002c, SR=ffffffff8400ffb3, BADVADDR=fac06b8
=====
```

Panics

Processing

Assembler macro **PANIC** is a machine language interface to the C language **panic()** function. It passes the address of the PANIC message in the call.

Function **panic()** simply calls **icmn_error** with arguments **CE_PANIC** and the message address.

For example, from **irix/kern/bsd/socket/uipc_socket.c**: `m = so->so_rcv.sb_mb;`

```
if (m == 0)
    panic("receive 1");
```

Function **icmn_error()** prints panic message and processes dump (see below).

Panic Example

```
<0>PANIC: CPU 4: receive 1
```

Stack Trace Example

```
>> ctrace 4
=====
STACK TRACE FOR CPU 4

1 syncreboot[./os/printf.c: 1385, 0xc0000000005fbd58]
2 icmn_err[./os/printf.c: 469, 0xc0000000005f9bc4]
3 panic[./os/printf.c: 665, 0xc0000000005fa2fc]
4 soreceive[./bsd/socket/uipc_socket.c: 923, 0xc0000000004b8e7c]
5 recvit[./bsd/socket/uipc_syscalls.c: 740, 0xc0000000004d6950]
6 recvfrom[./bsd/socket/uipc_syscalls.c: 467, 0xc0000000004d5fdc]
7 syscall[./os/trap.c: 2561, 0xc000000000586e1c]
8 sysstrap[./ml/LOCORE/sysstrap.s: 308, 0xc0000000003256e0]
  r0/zero:0000000000000000  r1/at:000000007fffffff  r2/v0:0000000000000044a
  r3/v1:ffffffffffffffff  r4/a0:0000000000000004  r5/a1:000000001003c000
  r6/a2:0000000000002000  r7/a3:0000000000000000  r8/a4:000000001005d11c
  r9/a5:000000007ffffcd4  r10/a6:000000001005d12c  r11/a7:0000000000000000
  r12/t0:000000001003c068  r13/t1:00000000337ccf55  r14/t2:000000001003c060
  r15/t3:000000001003c064  r16/s0:0000000003000000  r17/s1:0000000003000000
  r18/s2:0000000000000000  r19/s3:0000000000200edc  r20/s4:0000000000000002
  r21/s5:0000000006000000  r22/s6:0000000000000000  r23/s7:0000000000000004
  r24/t8:000000000008ecb3  r25/t9:00000000fabacc  r26/k0:0000000000000000
  r27/k1:000008050283002a  r28/gp:00000000fb601f0  r29/sp:000000007ffff1b80
  r30/s8:0000000000002000  r31/ra:00000000fb031a0  EPC:00000000fafba98
  CAUSE=1000002c, SR=ffffffffffa400ffb3, BADVADDR=fabacc
=====
```

Common Routines

`icmn_err()` Panic Processing

Called with flag (eg. `CE_PANIC`), "printf" format string, and operand values; for instance

```
icmn_err(CE_PANIC,
"%s\nIRIX Killed due to fatal memory ECC error.\n",buf)
```

Calls `icmn_err()` with `CE_PANIC` flag, passing format string and operand values as arguments

icmn_err() Panic Processing

1. Gets CPU number of panicing CPU.
2. Blocks all CPU interrupts.
3. Tests if not first CPU here; if not, spin wait.
4. Sets this CPU in panic state.
5. Formats panic message.
6. Increments panic level counter by 1.
7. If first time in panic processing (panic level counter == 1):
 1. Stops interval timer.
 2. Sets each other CPU's `panic_spin` flag and sends it an interrupt.
 3. Flushes console buffers.
 4. Prints panic error message to console.
 5. Formats and prints `syslog` message for `availmon`.
 6. Flushed console buffers
 7. Calls `dump_tlb()` to save TLB at `tlbdump_tlb`
 8. If this CPU had ECC error
Calls `panic_spin()` which selects another CPU to do dumping
(This CPU spins here - another CPU calls `dumpsys()` to finish dump)
 9. Tests if all TLBs are dumped; prints error message if not.
 10. Calls `sync_reboot()` to dump and restart the system. (*see below*)
 1. Locks `putbuf`
 2. Calls `dumpsys()` to write memory to disk (*see below*)
 3. NO RETURN from `dumpsys()`

8. If second time in panic processing(panic level counter == 2):
 1. Stops interval timer.
 2. Prints "DOUBLE PANIC" message.
 3. Updates `availmon` log.
 4. Flushes console buffers.
 5. Calls `mdboot(AD_HALT)` which calls `mprboot()` to attempt to reboot.
(NO RETURN from `mprboot`)
9. If third time in panic processing (panic level counter == 3):
 - o Calls `mdboot(AD_HALT)` which calls `mprboot()` to attempt to reboot.
(NO RETURN from `mprboot`)
10. If fourth (or more) panic processing (panic level counter > 3):
 - o Hangs in infinite spin loop.

syncboot() Processing

1. Locks `putbuf`
2. Calls `dumpsys()` to write memory to disk (*see below*)
3. NO RETURN from `dumpsys()`

dumpsys() Processing

1. Uses `setjmp` to save registers in `dumpregs`.
2. Save current process pointer in `dumpproc`.
3. Creates stack frame in preparation to call `real_dumpsys()`.
4. Flag this process to run in this (and only this) CPU
5. Uses `longjmp` to call `real_dumpsys()`; execute a NULL ASSERT is it ever returns
 1. Sets ~~spin lock~~ `7` (locks other CPUs in kernel).
 2. Panics or hangs system if improperly configured dump device.
 3. Sets dump in progress flag.
 4. Prints "Dumping to *device* at block *number*, space: *0xnumber* pages".
 5. Sets `physaddr` to first valid PFN in the system.
 6. Calls `dumpvmcore()` to dump memory to disk (*see below*).
 7. Clears dump in progress flag.
 8. Calls `mdboot()` which calls `mprboot()` to attempt to restart the system; hang CPU IF it returns.

SPL
Set Priority
Level

ie, disables interrupts

dumpvmcore() Processing

1. Disables ECC interrupts.
2. Calls `flush_cache()` to save cache memory to "primary" memory.
3. Sets up dump device header; writes it to disk.
Note - each section of the dump is separated by a descriptive dump header.
4. Writes `putbuf` data to disk.
5. Write `errbuf` data to disk.
6. Prints "Dumping low memory".
7. Calls `dump_page()` to dump `physaddr` to `physaddr+0x4000`:
Memory is compressed before writing to disk.
Bad or inaccessible pages are skipped.
Stops if "out of dump space".
Prints a "." for each "dump block" written.
8. Checks `dump_level` config variable - see "Dump level configuration" below.

9. if `dump_level >= 1`:
 1. Prints "Dumping static kernel pages...".
(Kernel code and compiled (static) data)
 2. Calls `dump_lowmem`:
On node zero - dumps from `physaddr+0x4000` to last page before `pfdat` table.
On non-zero nodes - dumps from 0 to last page before `pfdat` table.
Does NOT dump copies of kernel on non-zero nodes.
 3. Prints "Dumping dynamic kernel pages...".
 4. Calls `pdf_scan(SCAN_KERN_NONBULK, DUMP_SELECTED)`
(Kernel dynamic (malloc'd) memory, excluding system buffers and mbufs).
10. if `dump_level >= 2`:
 1. Prints "Dumping buffer pages..."
 2. Calls `pdf_scan(SCAN_KERN_BULK, DUMP_SELECTED)`
(Kernel dynamic (malloc'd) system buffer and mbuf memory).
11. if `dump_level >= 3`:
 1. Prints "Dumping remaining in-use pages..."
 2. Calls `pdf_scan(SCAN_INUSE, DUMP_INUSE)`
(Allocated memory, not yet dumped)..
12. if `dump_level >= 4`:
 1. Prints "Dumping free pages..."
 2. Calls `pdf_scan(SCAN_FREE, DUMP_FREE)`
(The rest of memory).
13. Flush un-written dump buffers to disk.
14. Prints "Updating dump header..."
Prints "Dump complete."
15. On SN0 - prints "System dump completed" to PROMLOG.

Dump Level Configuration

Dump Level

Configuration variable `dump_level` is set and displayed by `sysctl(1M)`

```
flurry(133): sysctl
  group: lockd (dynamically changeable)
    max_lockd_procs = 14768 (0x39b0)
    nlm_callback_retry = 3 (0x3)
{edited}
  group: misc (dynamically changeable)
    disable_bte = 0 (0x0)
{edited}
    dump_level = 4 (0x4)
{edited}
```

Dump level meanings

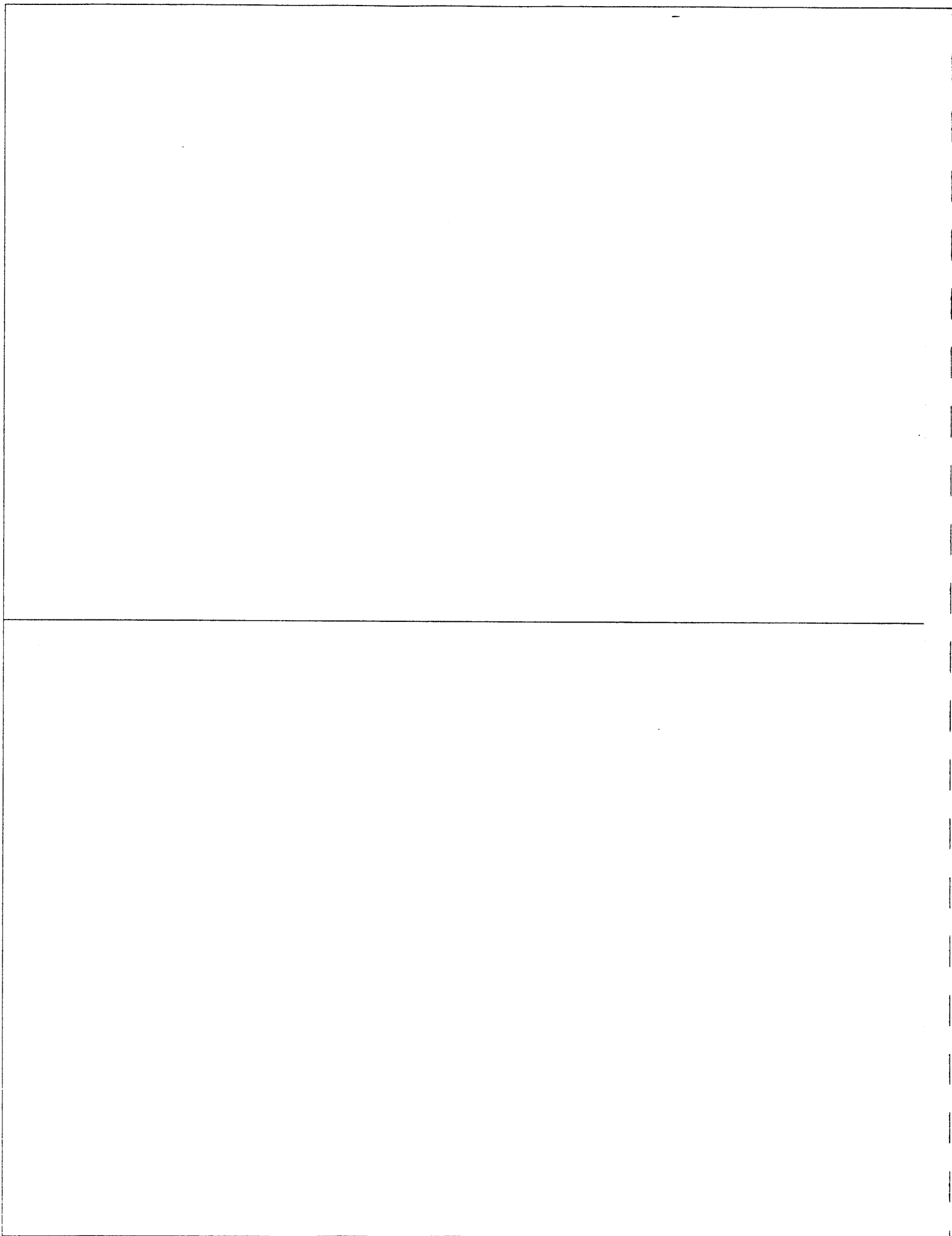
Dump routines repeatedly scan the kernel memory management table `pfdat`.

Each page is flagged as its use in the kernel.

Once a page is dumped, it is marked as "dumped" so it does not get dumped again.

Dumping proceeds **IN THIS ORDER**:

- **Level 1 or greater: kernel code, static data, and non-bulk dynamic data**
 - First valid PFN to first valid PFN+0x4000 (always dumped)
 - Kernel static memory (code and static data)
 - Node zero: first valid PFN+0x4000 to last page before `pfdat` table
 - Non-zero nodes: first valid PFN to last page before `pfdat` table
 - Kernel dynamic (malloc'd) memory excluding system buffers and mbufs
- **Level 2 or greater: kernel bulk (dynamic) data**
 - System buffers
 - mbufs (Message buffers)
- **Level 3 or greater: in-use memory**
 - Memory pages marked as allocated in the `pfdat` table
- **Level 4 or greater: free memory**
 - Memory pages marked as free in the `pfdat` table



Module 17: Bibliography

Bibliography

This bibliography contains suggested references organized by the following categories:

- BOOKS BY SGI EMPLOYEES (former or current)
- BOOKS
- ON-LINE DOCUMENTS
- TOOLS
 - Browsing Kernel Source Code
 - Browsing Executable Code (".c" files)
 - Browsing Memory
- TRAINING MATERIALS WEB PAGES
- INSTRUCTOR WEB PAGES (links to their class materials)
- ENGINEER WEB PAGES

17-1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

BOOKS BY SGI EMPLOYEES (former or current)

Scalable Shared-Memory Multiprocessing

by Daniel E. Lenoski, Wolf-Dietrich Weber, Dan Lenoski

(One of our Principle Engineers said: "...recommend the very scholarly well-written and complete introduction to NUMA-style architectures (including the concept of directories) written in part by our own Dan Lenoski when he was at Stanford. Dan then joined SGI and became the head of the SN0 project. (at least the first half of the book) should be *required reading* for anyone that teaches classes about our hardware... you might want to recommend it for the students.")

Unix Systems for Modern Architectures : Symmetric
Multiprocessing and Caching for Kernel Programmers
(Addison-Wesley Professional Computing Series)
by Curt Schimmel

Order books on-line from:

<http://www.amazon.com/exec/obidos/ISBN%3D1558603158/1963-2407536-869114>

17-2

22jul1998

TR-IKI rev 0.7b SGI Proprietary

BOOKS

The Magic Garden Explained
The Internals of Unix System V Release 4 : An Open Systems Design
Beryn Goodheart, James Cox / Paperback / Published 1994

Order books on-line from:
<http://www.amazon.com/exec/obidos/ISBN%3D1558603158/1963-2407536-869114>

ON-LINE DOCUMENTS

Documents / Technical Information Menu
<http://www-devtoolbox.engr.sgi.com/toolbox/documents/>

Hardware Developer Handbook, Release 2.0
<http://www-devtoolbox.engr.sgi.com/toolbox/hardware/hwHandbook/>

R10000 Microprocessor User's Manual -Version 2.0 -Copyright 1996, 1997, MIPS Technologies, Inc. -- 09 DEC 96
http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/HTML/t5.Ver.2.0.book_1.html

MIPS IV Instruction Set
http://coral.mti.sgi.com/arch/MIPS4_3.2/APP.book_1.html

R10000 Microprocessor User's Manual
http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/HTML/t5.Ver.2.0.book_1.html

IRIX 6.3/6.4 Migration Course
(Virtual Memory Overview)
http://snt.engr.sgi.com/webverter/cameron/6.3_6.4_Migration/CDROM/

(Memory, Swap, Tuning Information)
<http://catlady.engr.sgi.com/~saragon/CoCreate/customization/chap3.htm#sgiopt-2>

Pthreads Home Page
<http://www-devtoolbox.engr.sgi.com/toolbox/documents/pthreads/index.html>

IRIX 6.4 Device Driver Programming Guide
(Operating System Overview Information, mostly still accurate for 6.5)
<http://www-devtoolbox.engr.sgi.com/toolbox/documents/DevDriver/irix6.4/DD6.4toc.html>
How to download it
<http://www-devtoolbox.engr.sgi.com/toolbox/documents/DevDriver/irix6.4/html/>

Architecture Documentation Database

(SNO architecture specifics)
<http://b7.asd.sgi.com/doc/arch/>

including:

Lego System Specification
http://b7.asd.sgi.com/doc/arch/lego/sys_spec.book.doc.html

Link Level Protocol Specification
http://b7.asd.sgi.com/doc/arch/llp/llp_spec.book.doc.html

Lego Cache Coherence Protocol Specification
(including DIMMs <Directory Memory> Explanation and Block Diagrams)
http://b7.asd.sgi.com/doc/arch/coherence/coherence_spec.book.doc.html

ASD/NSD 1996 Next Generation Product Specification
http://b7.asd.sgi.com/doc/arch/prod_spec/ProdSpec.book.html

PCI-to-PCI Bridge Issues in Origin Systems
http://b7.asd.sgi.com/doc/arch/pci_to_pci/pci_to_pci.doc.html

Origin I/O Memory Model
http://b7.asd.sgi.com/doc/arch/io_mem_model/io_mem_model.doc.html

17-4.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

TOOLS

Browsing Kernel Source Code

cscope - see tutorial "IRIX source browsing"
<http://www.tng.cray.com/~mix/irix.html#Class-Materials>

Browsing Executable Code (".c" files)

elfdump

dwarfdump - see tutorial "IRIX source browsing"
<http://www.tng.cray.com/~mix/irix.html#Class-Materials>

Browsing Memory

IRIX Crash Command Set (icrash) - a web page listing icrash commands. Clicking on any of them gets you to giving a brief definition of that directive, and some examples.
<http://bits.csd.sgi.com/digest/saca/icrash/commands.html>

icrash - see tutorial "IRIX source browsing"
<http://www.tng.cray.com/~mix/irix.html#Class-Materials>

17-5

22jul1998

TR-IKI rev 0.7b SGI Proprietary

TRAINING MATERIALS WEB PAGES

IRIX Software Training
<http://www.tng.cray.com/~mix/irix.html>

INSTRUCTOR WEB PAGES (links to their class materials)

Howard Mundy's web page:
<http://www.tng.cray.com/~hlm>

Dave Wright's web page:
<http://www.tng.cray.com/~daw>

Cliff Wickmans's web page:
<http://www.tng.cray.com/~cpw>

Mike Conrad's web page:
<http://www.tng.cray.com/~conrad>

ENGINEER WEB PAGES

Chandler Lai's web page:
<http://chandler.csd.sgi.com/~clai2>

which includes:

IRIX 6.5 Support Readiness Information
<http://chandler.csd.sgi.com/~clai2/6-5info.html>

IRIX 6.5 Engineering Technical Information Overview Sessions
<http://chandler.csd.sgi.com/~clai2/ETIO-TACbeta.html>

(list of publications)
<http://chandler.csd.sgi.com/~clai2>

Appendix A: Origin2000 Support Processes For High End Systems



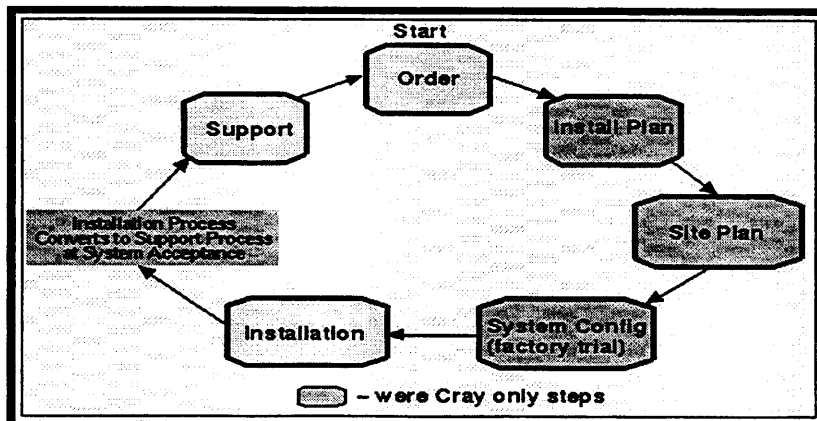
Purpose

- Communicate Cray Origin2000 system processes and procedures.
- Introduce the tools supporting these processes and procedures.

These processes has evolved over time and has been driven from key areas of need:

1. **A strong desire to avoid any unexpected surprises after the system has shipped.**
Most of these systems have an acceptance period with specific criteria which must be met before customer acceptance is completed and revenue flows.
2. **Feedback and process improvement.**
The general belief is that the closer to the factory a problem is identified and fixed, the less costly it is to fix.

Thus, it is important to provide input on how well installations go; then, we can go back and fix problems and improve processes.



A Cray Origin2000 system is defined as a high end Origin2000 or array of Origin2000 systems greater than 32 CPUs. Origin2000 systems which start out as configurations smaller than 32 CPUs and are upgraded to above 32 CPUs are then considered Cray Origin2000 systems and follow the procedures described below.

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix A-2.a

Getting Help

Please contact Dave Walls (walls@cray.com, +1-612-683-5352) regarding any questions on the material in this document.

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix A-3

Getting Cray domain accounts

Some of the tools and processes described below require Cray domain accounts.

Request CrayRealm and Training domain accounts

Appendix A-4

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Site Planning

Currently available site planning materials: (05/97)

Table A-1: Site Planning Materials

007-3452-xxx	<i>Site Preparation for Origin Family and Onyx2</i>
HR-04122	<i>CRAY Origin2000 or Onyx2 InfiniteReality Rack System</i>
10658642	<i>Origin2000 Rack Site Planning Packet</i>
10658643	<i>Challenge RAID Site Planning Packet</i>
10658644	<i>Operators System Console Site Planning Packet</i>

Planning packets can be requested by mailing site@cray.com or calling +1-715-726-2820. Additional information concerning power requirements and floor plan layouts is available at the [Cray Site Engineering Group's home page](#).

Power Requirements tool

Appendix A-5

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Installation Planning

A new tool has been created to aid Branch Support Managers (BSMs) in planning for the delivery, installation, and acceptance of large Origin2000 systems. This Installation Plan Manager (ipm) tool supports the writing of installation plans for all Origin2000 systems.

You are requested to use this ipm tool to write an install plan for all large Origin2000 systems (Origin2000 systems with more than 16 CPUs).

Completing the installation plan helps to ensure accurate delivery and success of your system installation and acceptance by the customer. Besides identifying the field account team responsible for this work, some of the other valuable information included in the plan is:

- customer's intended use of the system
- production hardware and software configurations
- installation timeline
- upgrade timeline
- acceptance timeline



System Registration

System serial number for Origin systems greater than 32 CPUs must be registered in the CRUISE database.

This registration provides these benefits:

1. Customer calls within the U.S. for these systems are routed to the Eagan-TAC instead of the Mountain View TAC.
2. Availmon data from this list of registered systems is used to calculate MTTI for large Origin systems.
3. Some traditional Cray customers still wish to use the CRInform tool to report software problems (SPRs). This CRUISE registration is necessary to validate their use of CRInform and report problems.

To register the system under CRUISE, send this information to cruise_reg@cray.com:

- Customer Name
- System Serial Number
- Support Branch in which the system is located.

Register Origin System Under CRUISE

Currently, this is the only use of CRUISE required for Origin systems.

System Serial Number

The ORIGIN system serial number needed to register the system in CRUISE is from the lower module located at the left end of the system as you face the front of the system.

- Serial number format is Knnnnnnn.
- The serial number is located on a white sticker at the back of the module/rack behind the power cord connection to the fan tray (left side). It is important that this physical module's serial number be used as the system serial number for tracking purposes.
 - This is not always the same as the Sales Order's serial number.
 - Some early systems require that the upper module's serial number be used instead. In these cases, the upper module had a lower valued serial number than the lower module.
- Software commands to obtain serial numbers are:

Command	Prints
<code>/usr/etc/amsysinfo</code>	system serial number
<code>sysinfo -vv cut -d' ' -f2</code>	all serial numbers (1st is system)
<code>hinv -vm</code>	serial numbers of all modules

Installation Reporting

In the U.S., a Clarify install ticket should be opened to track the status of the installation. This can be done through the ESCALL tool in Supportfolio or by using Clarify directly.

Information that should be included is:

- the install time
- any installation problems
- any hardware fallout

Resources are allocated to monitor this information and work with the appropriate groups to resolve the problems and ensure better future installations.

Initial Mainframe Hardware Install Reporting

Update your system's Clarify install ticket to report:

- hardware installation status
- time taken to install the hardware
- how well the initial hardware install went

A Initial Mainframe Install-HW form is not required for Origin systems.

<i>Origin2000 and Onyx2 Deskside And Rackmount Installation Instructions</i>	108-0155-xxx
<i>Origin2000 Power-On Diagnostics</i>	108-0161-xxx

Initial Mainframe Software Install Reporting

Update your system's Clarify install ticket to report:

- software installation status
- time taken to install the software
- how well the initial software install went

A Initial Mainframe Install-SW form is not required for Origin systems.

System Verification Program (svp)	Reference Guide 108-0165-xxx	FAQ
<i>FRU Analyzer Reference Guide</i>		108-0166-xxx
<i>ICRASH Reference Guide</i>		108-0167-xxx

Hardware & Software Installation Defects

Update your system's Clarify install ticket to report problems encountered in hardware and software installation processes.

Install Hardware Defect & Install Software Defect forms are not required for Origin systems.

System Failure Reporting

Once the system has been accepted, all known hardware or system failures requiring a reboot of the system should be reported. This information is used to determine Mean Time To Interrupt (MTTI) reliability metrics.

MTTI reliability metrics are used to:

- provide input back to the engineering groups on system stability
- identify areas for improvement.

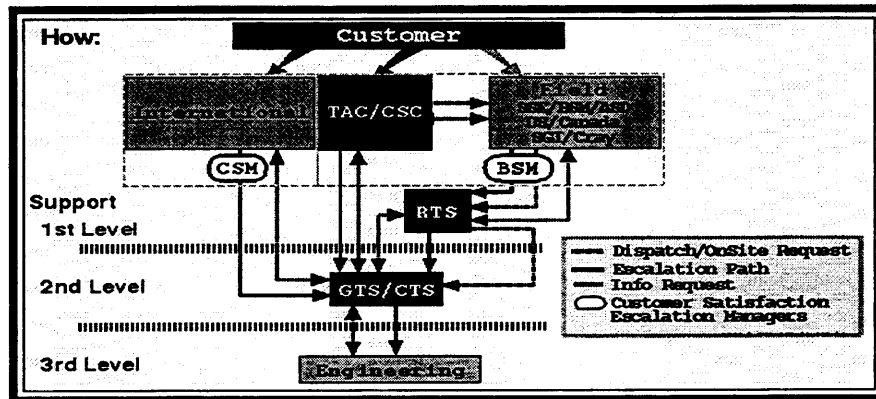
All, both U.S. and International, Origin 2000 sites should run Availmon if the customer allows it. Availmon is being used to capture failure data which is used to calculate system Mean Time To Interrupt (MTTI) metrics. Previously sites were being asked to report failure information in the Cruise database.

[AvailMon](#) [More Information](#)

Problem Escalation

One escalation process applies to all Origin systems, regardless of size. The next diagram, originated by Jack La Salle of RTS, summarizes this overall escalation model.

Figure A-0: Critical Problem Escalation



In North America, escalation of large Origin system problems to backbone support groups, GTS/CTS, follow standard GTS escalation procedures.

Escalation of large Origin system problems originating with SSEs in North America begin when they escalate to the RTS organization (1-888-800-4RTS). If Regional Technical Analyst (RTAs) requires further escalation, they escalate to GTS either through the GTS Escalation Pager (1-415-588-4826) for critical system down/customer satisfaction issues or through standard GTS escalation procedures for normal escalations.

More information:	GTS's Escalation Procedure & Information Guideline					
	RTS's Escalation Procedure for U.S. & Canada					RTA Location & Expertise
Home Pages	BSM/SSE	CSC	CTS	GTS	RTS	Egan & MV TAC SGI & Cray Logistics

GTS's Hotlist

Employees may monitor escalation status by viewing GTS's Hotlist. This Hotlist is the forum for escalating issues into Engineering.

Hot Site List **Hot Accounts** **Top Issues** **Hot Bugs**

The Cray Weekly Site Review (WSR) process and report is no longer used for escalating issues and problems for any Origin systems. Work is in process to merge these two escalation forums.

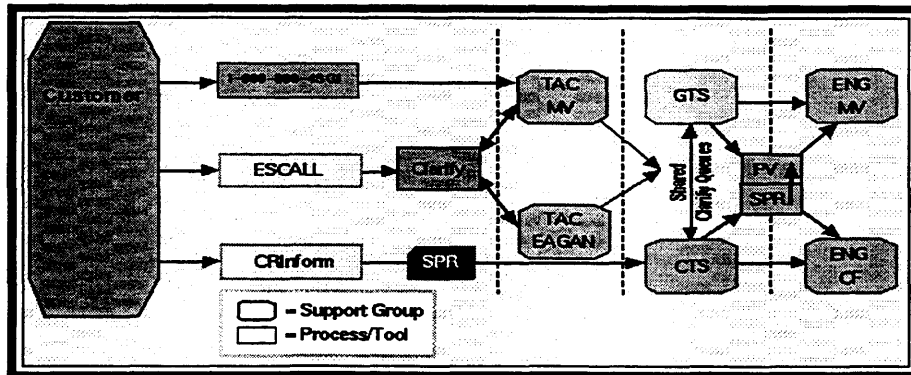
U.S. Escalation Model

High end Origin system customers have access to some traditional Cray processes and tools that have not yet been merged into a single set of processes and tools. This results in a need to deviate from the standard processes used for low end Origin systems. Processes and tools which necessitate deviation are:

- customer access to the CRInform tool
- the ability to submit SPRs in lieu of PVs
- high end U. S. region calls routed to the Eagan TAC.

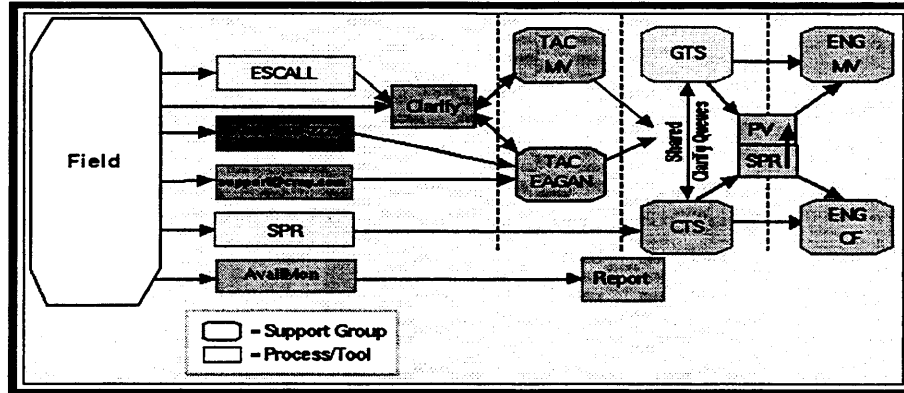
The next charts help explain problem flow through the organizations with these differences added.

Figure A-1: Cray Origin 2000 U.S. Customer's Problem Flow



As shown, the backline support team works on one set of queues to address all Origin 2000 escalations, regardless of system size. <

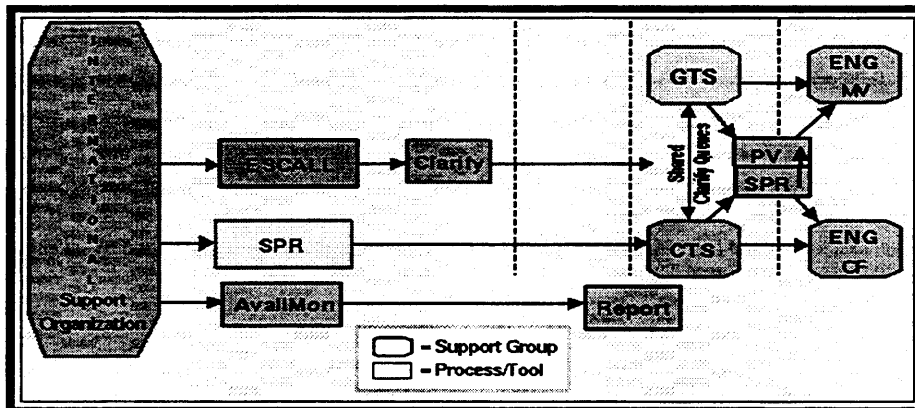
Figure A-2: Cray Origin 2000 U.S. Field's Problem Flow



Field in this diagram, includes SSEs, RTAs, and ASEs.

International Escalation Model

Figure A-3: Cray Origin 2000 International Problem Flow



International regions are expected to handle escalations locally, with existing processes. When it becomes necessary to escalate to GTS in the U.S., use the current escalation procedures defined for any other Irix based system. Both non-critical and critical escalations requires that a Clarify call be created. This can be done through the ESCALL tool in Supportfolio. The Clarify call that is created will be assigned to the appropriate support group in the U.S.

Problem Reporting

Clarify is the call routing tool to initiate requests for help for all Origin2000 systems in the U.S. and for escalating calls from International to the U.S. backline support groups.

- Critical (system down) problems are escalated via direct contact.
- Non-Critical problems are passed via Clarify.

Clarify is used to document all problems, hardware and software, critical and non-critical.

This supports a consistent process for getting help and escalating problems for all Origin2000 machines, regardless of size. Clarify directs escalations to correct backline support groups based on system registration information.

Report problems by any of these means:

- Use Clarify directly.
- Use Clarify indirectly by contacting the call center assigned to the customer.
- Use the Electronic Escalation CallLog (ESCALL) capability in Worldwide Customer Service Supportfolio OnLine.

Clarify		Launch
Supportfolio OnLine	Data Sheet	Launch
More information:	Supportfolio	Supportfolio OnLine
		Electronic Escalation CallLog (ESCALL)

Software Problem Reporting

All Cray Origin2000 support organizations are using PVs internally. Software Problems Reports (SPRs) go into the SPR database. Cray Origin2000 system SPRs automatically convert into PVs to support the MV and CF engineering organizations need to use existing tools.

The ProjectVision (PV) number will match the SPR number.

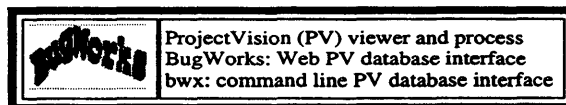
At customer request, field personnel can file SPRs to report software problems against Cray Origin2000 systems as an alternate to PVs.

Field personnel not familiar filing SPRs can initiate a Clarify call or ESCALL and request that a problem also be recorded as an SPR.

Using the SPR database allows customers to check on the status of their problems from problem report through patch (fix) availability through CRInform.

Long-term plans are to replace SPR and PV with a new SCOPUS-based tool (BugMaster) that can be used by all engineering organizations. BugWorks is planned to bridge to this future database from its current PV database.

Display SPR: PV: Example, try 706050.



Customer Communication

Customer Communication	CRSB (n/u)	Pipeline
	CRInform	Supportfolio On-line
	FN (some)	None
	Site prep docs	None

Customer communication mechanisms:

- Pipeline
- CRInform
- Field Notices and FYIs/FIBs/NPIs

Pipeline

The Pipeline publication:

- is the communication mechanism for topics related to Cray Origin2000 systems.
- is made available on the Supportfolio CD-ROM.
- will be sent to all Origin2000 customers.

Pipeline	home page	On-line Viewer
--------------------------	---------------------------	--------------------------------

Cray Inform (CRInform)

CRInform is an online information and problem-reporting service for SGI/Cray customers and employees.

Cray Origin2000 customers will be provided with an CRInform account. This CRInform account will be set up when the Origin system is registered under CRUISE.

CRInform provides customers with the capability to:

- track the status of software problems they have reported against their Cray Origin2000 system
- view Field Notices that have been written against Cray Origin2000 systems.

CRInform features:

- Report software problems (customers only)
- Request technical assistance (customers only); similar to ESCALL
- Search information repositories, including SPRs, Field Notices, Cray Service Bulletins, and Software Release Documents
- Order software, software updates, software fixes, and software publications
- Access the Publications and Training Catalogs
- Access customer bulletin boards
- Customize a user profile for receiving email notification when information of interest to you is available
- Read about new products

CRInform	Introduction	External & Internal home page
--------------------------	------------------------------	---

Field Notices and FYIs/FIBs/NPIs

A Field Notice (FN) is a SGI/Cray document that communicates technical or procedural information about SGI/Cray products to customers, employees, and third-party service providers. FNs are similar in function to SGI's Field Information Bulletins (FIBs).

Both Field Notices and FYIs/FIBs/NPIs will be used to communicate information concerning Cray Origin2000 systems.

Plans are to combine these two communication processes into one process. In the interim, to ensure all areas are covered:

- All Cray Origin2000 Field Notices will have an associated FYI, FIB, or NPI with similar information.
- **FYIs, FIBs and NPIs do not go to customers.** FYIs, FIBs, and NPIs which contain information that is to be sent to Cray customers, will be re-written as a Field Notice (FN) and distributed to field personnel that are registered to receive Cray Origin2000 field notices and to customers via CRInform.
- FYIs, FIBs, and NPIs that apply against Cray Origin2000 systems and contain information that will **not** be distributed to customers, will be mailed to field personnel registered to receive Cray Origin2000 Field Notices.

Format for these FYIs, FIBs, and NPIs re-issued as field notices will not be changed to Field Notice format.

Field personnel who need to receive all Origin2000 communications should register to receive:

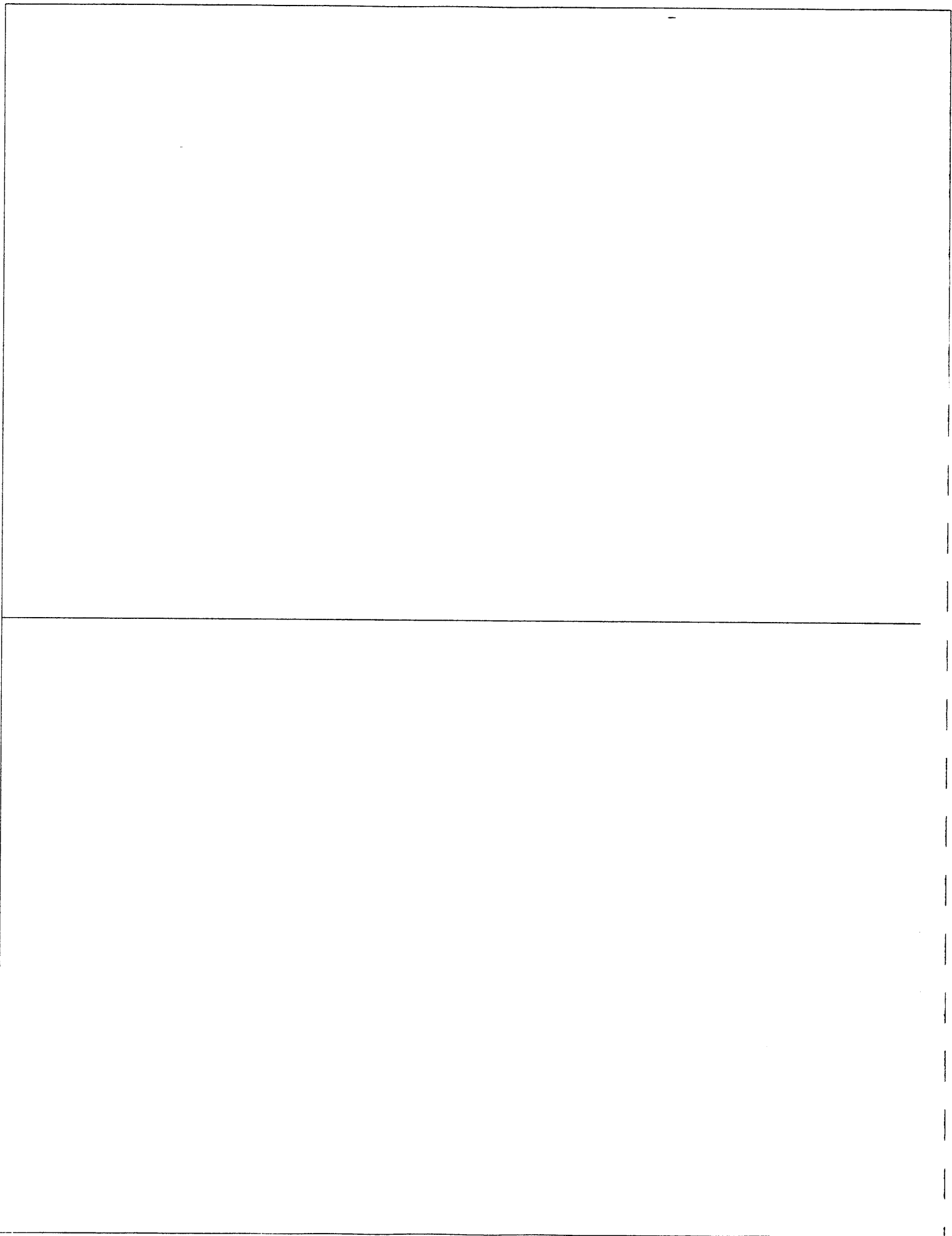
- Field Bulletin System (FBS) mailings
- Field Notice (FN) mailings

FBS mailings	join leave
FN mailings	Register More Information

Related Information

- Cray Origin2000 Support Tools and Planning
- Origin/Onyx2 Firmware information on the Lego Software home page
- Supporting documentation:

<i>Origin2000 Deskside Owner's Guide</i>	007-3453-xxx
<i>Onyx2 Deskside Workstation Owner's Guide</i>	007-3454-xxx
<i>Origin Vault Owner's Guide</i>	007-3455-xxx
<i>Origin2000 Rackmount Owner's Guide</i>	007-3456-xxx
<i>Onyx2 Rackmount Owner's Guide</i>	007-3457-xxx
<i>Internal Support Tools CD</i>	814-0640-001



Appendix B: CPU R10000 Overview

MIPS[®] R10000 Microprocessor Overview

Appendix B-1

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Instruction prefetch

Prefetching of instructions is a technique whereby the processor can request a cache block prior to the time it is actually needed. For example, assume the compiler is progressing sequentially through a segment of code. The compiler can make the assumption that this sequence will continue beyond the range of addresses available in the on-chip cache and issue a prefetch instruction which fetches the next block of instructions in the sequence and places them in the secondary cache. Therefore, when the processor requires the next sequence, the block of instructions exist in the secondary cache or a special instruction buffer as opposed to main memory and can be fetched by the processor at a much faster rate. If for some reason the block of instructions is not needed, the area in the secondary cache or the buffer is simply overwritten with other instructions.

Appendix B-2

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Out-of-order execution

In a typical pipelined processor which executes instructions *in-order*, each instruction depends on the previous instruction which produced its operands. Execution cannot begin until those operands become valid. If the operands required to execute a given instruction are not valid, the pipeline stalls until those operands become valid. Because instructions execute *in order*, stalls usually delay all subsequent instructions.

In an *in-order* superscalar machine where multiple instructions are fetched each cycle, several consecutive instructions can begin execution simultaneously if all of their corresponding operands are valid. However, the processor stalls at any instruction whose operands are not valid.

In an *out-of-order* superscalar machine each instruction is eligible to begin execution as soon as its operands become available regardless of the original instruction sequence. The hardware effectively re-arranges instructions in order to keep the various execution units busy. This process is called *dynamic issuing*.

Queuing structures

The R10000 Microprocessor contains three instruction queues. These queues dynamically issue instructions to the various execution units. Each queue uses instruction tags to track instructions in each execution pipeline stage. Each queue performs dynamic scheduling and can determine when the operands that each instruction needs are available. In addition, the queues determine the execution order based on the availability of the corresponding execution units. When the resources become available the queue releases the instruction to the appropriate execution unit.

Integer Queue

The integer queue contains 16 entries and issues instructions to the two integer arithmetic logic units (ALU). Integer instructions are written into empty queue entries and up to four entries may be written each cycle. Integer Instructions remain in the queue until being issued to an ALU.

Floating Point Queue

The floating point queue contains 16 entries and issues instructions to the floating-point adder and floating-point multiplier execution units. Floating point instructions are written into empty queue entries and up to four entries may be written each cycle. Instructions remain in the queue until being issued to an execution unit. The floating-point queue also contains multiple-pass sequencing logic for instructions such as the multiply-add. This instruction is dispatched first to the multiply unit, then passed directly to the adder unit.

Address Queue

The address queue issues instructions to the Load-Store unit and contains 16 entries. The queue is organized as a circular FIFO (first-in first-out) buffer. Instructions can be issued in any order, but must be written to or removed from the queue in sequential order. Up to four instructions can be written every cycle. The FIFO maintains the programs original instruction sequence so that memory address dependencies may be computed easily.

An issued instruction may fail to complete because of a memory dependency, a cache miss, or a resource conflict. In these cases the address queue must re-issue the instruction until it is completed.

Execution Units

The R10000 Microprocessor contains five execution units which operate independently of one another. There are two integer arithmetic logic units (ALU), two primary floating-point units, (including two secondary FP units which handle long-latency instructions such as divide and square root), and a load/store unit for address calculation.

Integer ALUs

There are two integer ALUs in the R10000 microprocessor defined as ALU1 and ALU2. Both ALUs perform standard add, subtract, and logical operations. ALU1 handles all branch and shift instructions, while ALU2 handles all multiply and divide operations using iterative algorithms.

Floating-Point units

The R10000 Microprocessor contains two primary floating point units. The adder unit handles add operations and the multiply unit handles multiply operations. In addition, two secondary floating point units exist (not shown in block diagram) which handle long-latency operations such as divide and square root.

Load/Store unit and the TLB

The Load/Store unit consists of the address queue, address calculation unit, translation lookaside buffer (TLB), address stack, store buffer, and primary data cache. The Load/Store unit performs load, store, prefetch, and cache instructions.

All load or store instructions begin with a 3-cycle sequence which issues the instruction, calculates its virtual address, and translates the virtual address to physical. The address is translated only once during the operation. The data cache is accessed and the required data transfer is completed provided there was a primary data cache hit.

If there is a cache miss, or if the necessary shared register ports are busy, the data cache and data cache tag access must be repeated after the data is obtained from either the secondary cache or main memory.

The Cray Origin2000 TLB contains 128 entries and translates virtual addresses to physical addresses. The virtual address can originate from either the address calculation unit or the program counter (PC).

Secondary Cache Controller

Secondary cache support for the R10000 microprocessor is provided by an internal secondary cache controller with a dedicated secondary cache port. A dedicated 128-bit bus transfers data at the 200 MHz internal operating frequency of the R10000 CPU, yielding a maximum secondary cache data transfer rate of 3.2 GBytes/second. The R10000 microprocessor also provides a 64-bit system interface data bus.

The secondary cache is implemented as two-way set associative. Maximum cache size is 16 MBytes. Minimum cache size is 512 KBytes. Transfer width is 128 bits, or (4) 32-bit words. Consecutive cycles are used to transfer larger blocks of data.

System Interface

The system interface of the R10000 microprocessor provides a gateway between the R10000 and its associated secondary cache, and the rest of the computer system. The system interface operates at the frequency of *SysCik* being supplied to the processor. The programmability of the system interface allows for clock speeds of 200, 133, 100, 80, 67, 57, and 50 MHz. All system interface outputs, as well as all inputs, are clocked on the rising edge of *SysCik*, allowing the system interface to run at the highest possible clock frequency.

In most microprocessor systems only one system transaction can occur at any given time. The R10000 microprocessor supports a split-level bus transaction protocol. Split-transaction allows additional processor and external requests to be issued while waiting for a previous response. A maximum of four outstanding transactions at any given time are supported.

R10000 Branch Unit

The branch unit of the R10000 microprocessor can decode and execute one branch instruction per cycle. A branch bit is appended to each instruction during instruction decode. These bits are used to locate branch instructions in the instruction fetch pipeline.

The path a branch will take is predicted using a *branch history* RAM. This RAM keeps track of how often each particular branch was taken in the past. The code is updated whenever a final branch decision is made.

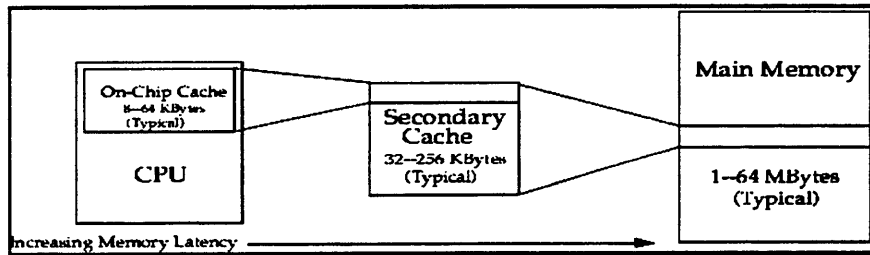
Any instruction fetched after a branch instruction is *speculative*, meaning that it is not known at the time these instructions are fetched whether or not they will be completed. The R10000 microprocessor allows up to 4 outstanding branch predictions which can be resolved in any order.

Special on-chip *branch stack* circuitry contains an entry for each branch instruction being speculatively executed. Each entry contains the information needed to restore the processor's state if the speculative branch is predicted incorrectly. The branch stack allows the processor to restore the pipeline quickly and efficiently when a branch miss-prediction occurs.

Branch instruction problem

All computer programs contain branch instructions. Some branches are unconditional, meaning that the program flow is always interrupted as soon as the branch instruction is executed. Other branches are conditional, meaning that the branch is taken only if certain conditions are met. Program flow interruption is inherent to all computer software and the microprocessor hardware has little choice but to deal with branches in the most efficient way possible.

When a branch is taken, the new address at which the program is to resume may or may not reside in the secondary cache. The latency is increased depending on where the new instruction block is located. Since the access times of the main memory and secondary cache are far greater than the on-chip cache, as shown in the below figure, branching can often degrade processor performance.



The branching problem is further compounded in super-scalar machines where multiple instructions are fetched every cycle and progress through stages of a pipeline toward execution. At any given time, depending on the size of the pipeline, numerous instructions can be in various stages of execution. When a conditional branch instruction is executed it is not known until many cycles later when the instruction is actually executed whether or not the branch should have been taken.

Implementation of branching is an important architectural problem. To improve performance many current architectures

incorporate branch prediction circuitry, which can be implemented in a number of ways.

Branch prediction

Since branch instructions interrupt the pipeline flow, branch prediction schemes are needed to minimize the number of interruptions. Branches occur frequently, averaging about one out of every six instructions. In super-scalar architectures where more than one instruction at a time is fetched, branch prediction becomes increasingly important. For example, in a four-way super-scalar architecture, where four instructions per cycle are fetched, a branch instruction can be encountered every other clock.

Most branch prediction schemes use algorithms which keep track of how a conditional branch instruction behaved the last time it was executed. For example, if the branch history circuit shows that the branch was taken the last time the instruction was executed, the assumption could be made that it will be taken again. A hardware implementation of this assumption would mean that the program would vector to the new target address and that all subsequent instruction fetches would occur at the new address. The pipeline now contains a conditional branch instruction fetched from some address, and numerous instructions fetched afterward from some other address. Therefore, all instructions fetched between the time the branch instruction is fetched and the time it is executed are said to be *speculative*. That is, it is not known at the time they are fetched whether or not they will be completed. If the branch was predicted incorrectly, the instructions in the pipeline must be aborted.

Appendix C: region.c source file

Appendix D: sbd.h

```
=====
sbd.h header file
/hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix/kern/sys/sbd.h
=====

#ifndef __SYS_SBD_H__
#define __SYS_SBD_H__

/*****
 *
 *      Copyright (C) 1990, Silicon Graphics, Inc.
 *
 * These coded instructions, statements, and computer programs contain
 * unpublished proprietary information of Silicon Graphics, Inc., and
 * are protected by Federal copyright law. They may not be disclosed
 * to third parties or copied or duplicated in any form, in whole or
 * in part, without the prior written consent of Silicon Graphics, Inc.
 *
 *****/

/* Copyright (C) 1986, MIPS Computer Systems */
/* Copyright (c) 1984 AT&T */
/* All Rights Reserved */

/* THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */
/* The copyright notice above does not evidence any */
/* actual or intended publication of such source code. */

#define $Revision: 3.112 $

#include <sys/mips_addrspace.h>

#if TFP
#include "sys/TFP.h"
#elif BEAST
#include "sys/beast.h"
#else /* R4000 || R10000 */

/*
 * Chip definitions for R3000 and R4000

```

```

*
* constants for coprocessor 0
*/
/*
* Exception vectors
*
* UT_VEC points to compatibility space in 64 bit R4000 systems. This is
* where the architecture spec defines it to be (and is needed so things
* like stack backtrace work). However, there are places in the kernel
* that we use UT_VEC for cache flush calls and the cache flush routine
* then tries to do IS_KSEG0(addr) which fails when it shouldn't. In these
* case, K0_UT_VEC should be used.
*/
#define SIZE_EXCVEC 0x80 /* Size (bytes) of an exc. vec */
#define UT_VEC COMPAT_K0BASE /* utlbmiss vector */
#define K0_UT_VEC K0BASE
#define R_VEC (COMPAT_K1BASE+0x1fc00000) /* reset vector */

#if R4000 || R10000
#define XUT_VEC (COMPAT_K0BASE+0x80) /* extended address tlbmiss */
#define ECC_VEC (COMPAT_K0BASE+0x100) /* Ecc exception vector */
#define E_VEC (COMPAT_K0BASE+0x180) /* Gen. exception vector */
#endif

#if R4000
#define MINCACHE 0x20000 /* 128 k */
#define MAXCACHE 0x400000 /* 4M */
#define R4K_MAXCACHELINESIZE 128 /* max r4k scache line size */
#define R4K_MAXPCACHESIZE 0x8000 /* max r4k primary cache size */
#if _PAGESZ > R4K_MAXPCACHESIZE
#define CACHECOLORSIZE 1
#else
#define CACHECOLORSIZE (R4K_MAXPCACHESIZE/NBPP)
#endif
#define CACHECOLORMASK (CACHECOLORSIZE - 1)

#if _PAGESZ >= R4K_MAXPCACHESIZE
#define CACHECOLORSHIFT 0
#else
#if _PAGESZ == 16384
#define CACHECOLORSHIFT 1
#else
#define CACHECOLORSHIFT 0
#endif
#endif
#endif

```

Appendix D-0.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

#define CACHECOLORSHIFT 3
#else
#ifdef _KERNEL
<<BOMB -- need define for unanticipated page size >>
#endif /* _KERNEL */
#endif /* !4096 */
#endif /* !16384 */
#endif /* ! >= R4K_MAXPCACHESIZE */
#endif /* R4000 */

#if R10000
#ifndef R4000
#define MINCACHE 0x80000 /* 512 k */
#endif /* R4000 */

#ifdef R4000
#undef MAXCACHE
#endif /* R4000 */
#define MAXCACHE 0x1000000 /* 16M */

#define R10K_MAXCACHELINESIZE 128 /* max r10k scache line size */
#define R10K_MAXPCACHESIZE 0x8000 /* max r10k primary cache size */
#ifndef R4000
#if _PAGESZ > R10K_MAXPCACHESIZE
#define CACHECOLORSIZE 1
#else
#define CACHECOLORSIZE (R10K_MAXPCACHESIZE/NBPP)
#endif
#define CACHECOLORMASK (CACHECOLORSIZE - 1)

#if _PAGESZ >= R10K_MAXPCACHESIZE
#define CACHECOLORSHIFT 0
#else
#if _PAGESZ == 16384
#define CACHECOLORSHIFT 1
#else
#if _PAGESZ == 4096
#define CACHECOLORSHIFT 3
#else
#define CACHECOLORSHIFT 0
#endif
#endif
#endif
#ifdef _KERNEL
<<BOMB -- need define for unanticipated page size >>
#endif
#endif /* !4096 */
#endif /* !16384 */

```

Appendix D-0.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

#endif /* ! >= R10K_MAXPCACHESIZE */
#endif /* R4000 */
#endif /* R10000 */

/*
 * TLB size constants
 */

#if R10000 && R4000
#define R10K_NTLBENTRIES 64
#define R4K_NTLBENTRIES 48
#define MAX_NTLBENTRIES R10K_NTLBENTRIES
#ifdef _KERNEL
#ifdef _LANGUAGE_C
extern int ntlbentries;
#define NTLBENTRIES ntlbentries
#endif /* _LANGUAGE_C */
#endif /* _KERNEL */

#else /* R10000 && R4000 */

#if R10000
#define NTLBENTRIES 64
#endif
#if R4000
#define NTLBENTRIES 48
#endif
#define MAX_NTLBENTRIES NTLBENTRIES
#endif /* R10000 && R4000 */

#ifdef MAPPED_KERNEL
#define NKMAPENTRIES 1
#define KMAP_INX 1
#else
#ifdef MH_R10000_SPECULATION_WAR
#define NKMAPENTRIES 2
#else
#define NKMAPENTRIES 0
#endif /* MH_R10000_SPECULATION_WAR */
#endif

#if _PAGESZ == 4096
#define NWIREENTRIES (8 + NKMAPENTRIES) /* WAG for now */
#endif

```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.c

```

#if _PAGESZ == 16384
#define NWIREENTRIES (6 + NKMAPENTRIES) /* WAG for now */
#endif
#define TLBWIREDBASE (1 + NKMAPENTRIES)
#define TLBRANDBASE NWIREENTRIES

#if R10000 && R4000
#define R10K_NRANDOMENTRIES (R10K_NTLBENTRIES - NWIREENTRIES)
#define R4K_NRANDOMENTRIES (R4K_NTLBENTRIES - NWIREENTRIES)
#define MAX_NRANDOMENTRIES R10K_NRANDOMENTRIES
#ifdef _KERNEL
#ifdef _LANGUAGE_C
extern int nrandomentries;
#define NRANDOMENTRIES nrandomentries
#endif /* _LANGUAGE_C */
#endif /* _KERNEL */

#else /* R10000 && R4000 */

#define NRANDOMENTRIES (NTLBENTRIES-NWIREENTRIES)
#define MAX_NRANDOMENTRIES NRANDOMENTRIES

#endif /* R10000 && R4000 */

#define TLBFLUSH_NONPDA TLBWIREDBASE /* flush all tlbs except PDA + kernel
                                     text mappings */
#define TLBFLUSH_NONKERN (TLBWIREDBASE+TLBKSLOTS) /* all but PDA/UPG/KSTACK */
#define TLBFLUSH_RANDOM TLBRANDBASE /* flush all random tlbs */

#define TLBINX_PROBE 0x80000000

#if R4000 || R10000
#if _PAGESZ == 4096
#define TLBHI_VFNSHIFT 12
#define TLBHI_VFNMASK _S_EXT_(0xffffe000)
#endif
#if _PAGESZ == 16384
#define TLBHI_VFNSHIFT 14
#define TLBHI_VFNMASK _S_EXT_(0xffff8000)
#endif
#endif
#if _MIPS_SIM != _ABI64
#define TLBHI_VFNZEROFILL 0
#else
#define TLBHI_VFNZEROFILL 0x3fffff0000000000

```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.d

```

#endif
#define TLBHI_VPNZMASK TLBHI_VPNMASK /* As named in the arch. spec.*/
#define TLBHI_VPN2SHIFT (TLBHI_VPNSHIFT+1)
#define TLBHI_PIDMASK 0xff
#define TLBHI_PIDSHIFT 0
#define TLBHI_NPID 255 /* 255 to fit in 8 bits */

#if defined(R10000) && (! defined(R4000)) && (_MIPS_SIM == _ABI64)
#define TLBLO_PFNMASK 0x3fffffff /* 28 bits for pfn */
#else /* defined(R10000) && (! defined(R4000)) && (_MIPS_SIM == _ABI64) */
#define TLBLO_PFNMASK 0x3fffffff
#endif /* defined(R10000) && (! defined(R4000)) && (_MIPS_SIM == _ABI64) */
#define TLBLO_PFNSHIFT 6
#define TLBLO_CACHMASK 0x38 /* cache coherency algorithm */
#define TLBLO_CACHSHIFT 3
#define TLBLO_UNCACHED 0x10 /* not cached */
#if _RUN_UNCACHED
#define TLBLO_NONCOHRNT TLBLO_UNCACHED
#define TLBLO_EXL TLBLO_UNCACHED
#define TLBLO_EXLWR TLBLO_UNCACHED
#else
#define TLBLO_NONCOHRNT 0x18 /* Cacheable non-coherent */
#define TLBLO_EXL 0x20 /* Exclusive */
#define TLBLO_EXLWR 0x28 /* Exclusive write */
#endif
#ifdef R10000
#define TLBLO_UNCACHED_ACC 0x38 /* Uncached Accelerated */
#endif /* R10000 */
#define TLBLO_D 0x4 /* writeable */
#define TLBLO_V 0x2 /* valid bit */
#define TLBLO_G 0x1 /* global access bit */

/*
 * TLBLO Uncached attributes field.
 */
#ifdef R10000
#define TLBLO_UATTRMASK 0xC000000000000000
#define TLBLO_UATTRSHIFT 62
#endif

#define TLBRAND_RANDOMMASK 0x3f
#define TLBRAND_RANDSHIFT 0

#define TLBWIREWIREDMASK 0x3f

```

Appendix D-0.e

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

#define TLBCTXT_BASEMASK 0xff800000
#define TLBCTXT_BASESHIFT 23
#define TLBCTXT_VPNMASK 0x7ffff0
#define TLBCTXT_VPNNORMALIZE 9
#define TLBCTXT_VPNSHIFT 4

#ifdef R10000
#define TLBEXTCTXT_BASEMASK 0xffffffe000000000
#define TLBEXTCTXT_BASESHIFT 37
#define TLBEXTCTXT_VPNMASK 0x7fffffff0
#define TLBEXTCTXT_REGIONMASK 0x0000001800000000
#define TLBEXTCTXT_REGIONSHIFT 27
#else /* R10000 */
#define TLBEXTCTXT_BASEMASK 0xffffffe000000000
#define TLBEXTCTXT_BASESHIFT 31
#define TLBEXTCTXT_VPNMASK 0x7fffffff0
#define TLBEXTCTXT_REGIONMASK 0x0000000180000000
#define TLBEXTCTXT_REGIONSHIFT 31
#endif /* R10000 */

#define TLBPGMASK_4K 0x0
#define TLBPGMASK_16K 0x0006000
#define TLBPGMASK_64K 0x001e000
#define TLBPGMASK_4M 0x07fe000
#define TLBPGMASK_16M 0x1ffe000

#if _PAGESZ == 4096
#define TLBPGMASK_MASK TLBPGMASK_4K
#endif
#if _PAGESZ == 16384
#define TLBPGMASK_MASK TLBPGMASK_16K
#endif

#endif /* R4000 || R10000 */

/*
 * Status register
 */

#ifdef R10000
#define SR_CUMASK 0x70000000 /* coproc usable bits */
#endif
#define SR_CU3 SR_XX

```

Appendix D-0.f

22jul1998

TR-IKI rev 0.7b SGI Proprietary


```

#endif /* R4000 */
#else
#define SR_CUMASK 0xf0000000 /* coproc usable bits */
#define SR_CU3 0x80000000 /* Coprocessor 3 usable */
#endif /* R10000 */

#define SR_XX 0x80000000 /* Enable Mips 4 inst. execution */
#define SR_CU2 0x40000000 /* Coprocessor 2 usable */
#define SR_CU1 0x20000000 /* Coprocessor 1 usable */
#define SR_CU0 0x10000000 /* Coprocessor 0 usable */

/* Diagnostic status bits */

#if R4000 || R10000
#define SR_SR 0x00100000 /* soft reset occurred */
#define SR_CH 0x00040000 /* Cache hit for last 'cache' op */
#endif
#define R10000
#define SR_NMI 0x00080000 /* NMI bit */
/* There is no CE bit on R10000 */

#endif /* R10000 */
#ifdef R4000
#define SR_CE 0x00020000 /* Create ECC */
#endif /* R4000 */
#define SR_DE 0x00010000 /* ECC of parity does not cause error */
#endif /* R4000 || R10000 */

#define SR_TS 0x00200000 /* TLB shutdown */
#define SR_BEV 0x00400000 /* use boot exception vectors */

/*
 * Interrupt enable bits
 * (NOTE: bits set to 1 enable the corresponding level interrupt)
 */
#if IP32
/*
 * Moosehead has different status register bit assignments and
 * uses an external interrupt controller. Only softints and
 * count/compare go directly to SR, CRIME controls all other
 * external interrupt sources in the system.
 */
#define SR_IMASK 0x0000ff00 /* Interrupt mask */
#define SR_IMASK8 0x00000000 /* mask level 8 */
#define SR_IMASK7 0x00008000 /* mask level 7 */
#define SR_IMASK6 0x00008400 /* mask level 6 */

```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.g

```

#define SR_IMASK5 0x00008400 /* mask level 5 */
#define SR_IMASK4 0x00008400 /* mask level 4 */
#define SR_IMASK3 0x00008400 /* mask level 3 */
#define SR_IMASK2 0x00008400 /* mask level 2 */
#define SR_IMASK1 0x00008600 /* mask level 1 */
#define SR_IMASK0 0x00008700 /* mask level 0 */
#else
#define SR_IMASK 0x0000ff00 /* Interrupt mask */
#define SR_IMASK8 0x00000000 /* mask level 8 */
#define SR_IMASK7 0x00008000 /* mask level 7 */
#define SR_IMASK6 0x0000c000 /* mask level 6 */
#define SR_IMASK5 0x0000e000 /* mask level 5 */
#define SR_IMASK4 0x0000f000 /* mask level 4 */
#define SR_IMASK3 0x0000f800 /* mask level 3 */
#define SR_IMASK2 0x0000fc00 /* mask level 2 */
#define SR_IMASK1 0x0000fe00 /* mask level 1 */
#define SR_IMASK0 0x0000ff00 /* mask level 0 */
#endif /* IP32 */

#define SR_IBIT8 0x00008000 /* bit level 8 */
#define SR_IBIT7 0x00004000 /* bit level 7 */
#define SR_IBIT6 0x00002000 /* bit level 6 */
#define SR_IBIT5 0x00001000 /* bit level 5 */
#define SR_IBIT4 0x00000800 /* bit level 4 */
#define SR_IBIT3 0x00000400 /* bit level 3 */
#define SR_IBIT2 0x00000200 /* bit level 2 */
#define SR_IBIT1 0x00000100 /* bit level 1 */

#if R4000 || R10000
/* SR_RP is undefined on R10000 - should be 0 */
#ifdef R4000
#define SR_RP 0x08000000 /* enable reduced-power operation */
#endif
#define SR_FR 0x04000000 /* enable additional fp registers */
#define SR_RE 0x02000000 /* reverse endian in user mode */

#define SR_KX 0x00000080 /* extended-addr TLB vec in kernel */
#define SR_SX 0x00000040 /* xtended-addr TLB vec supervisor */
#define SR_UX 0x00000020 /* xtended-addr TLB vec in user mode */
#define SR_KSU_MSK 0x00000018 /* 2 bit mode: 00b=>k, 10b=>u */
#define SR_KSU_USR 0x00000010 /* 2 bit mode: 00b=>k, 10b=>u */
#define SR_KSU_KS 0x00000008 /* 0-->kernel 1-->supervisor */
#define SR_ERL 0x00000004 /* Error level, 1=>cache error */

```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.h

```

#define SR_EXL 0x00000002 /* Exception level, 1=>exception */
#define SR_IE 0x00000001 /* interrupt enable, 1=>enable */
#define SR_IEC SR_IE /* compat with R3000 source */
#define SR_PREVMODE SR_KSU_MSK /* previous kernel/user mode */
#define SR_PAGESIZE 0 /* No pagesize bits in SR */
#define SR_DM 0 /* No FP Debug Mode bits in SR */
#define SR_DEFAULT 0 /* Bits to preserve in SR */
#if R10000
#define SR_KERN_SET SR_KADDR|SR_UXADDR /*Bits to set in SR for kernel mode*/
#else /* R10000 */
#define SR_KERN_SET SR_KADDR /* Bits to set in SR for kernel mode*/
#endif /* R10000 */
#define SR_KERN_USRKEEP 0 /* Bits to keep in SR from user mode*/

/*
 * SR_KADDR defines the desired state of the kernel address mode bit
 * in CO_SR, if such bits exist. We could actually enable SR_KX when
 * compiled under 32-bit compilers, though there is no real reason to do so.
 */
#if MIPS_SIM == _ABI64
#define SR_KADDR SR_KX /* kernel 64 bit addressing */
#define SR_UXADDR SR_UX /* user ext. addressing and opcodes */
#else
#define SR_KADDR 0 /* kernel 32 bit addressing */
#define SR_UXADDR 0 /* no user ext. address/opcodes */
#endif
#endif /* R4000 || R10000 */

#define SR_IMASKSHIFT 8

#if IP32
#define SR_CRIME_INT_OFF 0xffffbfff
#define SR_CRIME_INT_ON 0x00000400
#endif

#if IP20 || IP22 || IP28 || IP32 || IPMHSIM
/*
 * The following value is used as a flag to indicate whether
 * a status register value is saved in the pda. This is for
 * assertions that check for nested spsemahi calls. This value
 * can be any bit that does not conflict with the mips processor
 * level nor (on the IP6 with the lio mask value).
 */
#define OSPL_SPDBG 0x00000400

```

Appendix D-0.i

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

#endif /* IP20 || IP22 || IP28 || IP32 || IPMHSIM */

/*
 * Cause Register
 */
#define CAUSE_BD 0x80000000 /* Branch delay slot */
#define CAUSE_CEMASK 0x30000000 /* coprocessor error */
#define CAUSE_CESHIFT 28

/* Interrupt pending bits */
#define CAUSE_IP8 0x00008000 /* External level 8 pending */
#define CAUSE_IP7 0x00004000 /* External level 7 pending */
#define CAUSE_IP6 0x00002000 /* External level 6 pending */
#define CAUSE_IP5 0x00001000 /* External level 5 pending */
#define CAUSE_IP4 0x00000800 /* External level 4 pending */
#define CAUSE_IP3 0x00000400 /* External level 3 pending */
#define CAUSE_SW2 0x00000200 /* Software level 2 pending */
#define CAUSE_SW1 0x00000100 /* Software level 1 pending */

#define CAUSE_IPMASK 0x0000FF00 /* Pending interrupt mask */
#define CAUSE_IPSHIFT 8

#if R4000 || R10000
#define CAUSE_EXCMASK 0x0000007C /* Cause code bits */
#endif
#define CAUSE_EXCSHIFT 2

#define CAUSE_FMT "\20\40BD\36CE1\35CE0\20IP8\17IP7\16IP6\15IP5\14IP4\13IP3\12SW2\11SW1\1INT"

#define setsoftclock() siron(CAUSE_SW1)
#define setsoftnet() siron(CAUSE_SW2)
#define acksoftclock() siroff(CAUSE_SW1)
#define acksoftnet() siroff(CAUSE_SW2)

/* Cause register exception codes */
#define EXC_CODE(x) ((x)<<2)

/* Hardware exception codes */
#define EXC_INT EXC_CODE(0) /* interrupt */
#define EXC_MOD EXC_CODE(1) /* TLB mod */
#define EXC_RMISS EXC_CODE(2) /* Read TLB Miss */
#define EXC_WMISS EXC_CODE(3) /* Write TLB Miss */

```

Appendix D-0.j

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

#define      EXC_RADE      EXC_CODE(4)      /* Read Address Error */
#define      EXC_WADE      EXC_CODE(5)      /* Write Address Error */
#define      EXC_IBE      EXC_CODE(6)      /* Instruction Bus Error */
#define      EXC_DBE      EXC_CODE(7)      /* Data Bus Error */
#define      EXC_SYSCALL   EXC_CODE(8)      /* SYSCALL */
#define      EXC_BREAK    EXC_CODE(9)      /* BREAKpoint */
#define      EXC_II       EXC_CODE(10)     /* Illegal Instruction */
#define      EXC_CPU      EXC_CODE(11)     /* CoProcessor Unusable */
#define      EXC_OV       EXC_CODE(12)     /* Overflow */
#if R4000 || R10000
#define      EXC_TRAP     EXC_CODE(13)     /* Trap exception */
#define      EXC_VCEI     EXC_CODE(14)     /* Virt. Coherency on Inst. fetch */
#define      EXC_FPE      EXC_CODE(15)     /* Floating Point Exception */
#define      EXC_WATCH    EXC_CODE(23)     /* Watchpoint reference */
#define      EXC_VCED     EXC_CODE(31)     /* Virt. Coherency on data read */
#endif

/* software exception codes */
#define      SEXC_SEGV    EXC_CODE(32)     /* Software detected seg viol */
#define      SEXC_RESCHE  EXC_CODE(33)     /* resched request */
#define      SEXC_PAGEIN  EXC_CODE(34)     /* page-in request */
#define      SEXC_CPU     EXC_CODE(35)     /* coprocessor unusable */
#define      SEXC_BUS     EXC_CODE(36)     /* software detected bus error */
#define      SEXC_KILL    EXC_CODE(37)     /* bad page in process (vfault) */
#define      SEXC_WATCH   EXC_CODE(38)     /* watchpoint (vfault) */
#if R4000
#define      SEXC_EOP     EXC_CODE(39)     /* end-of-page trouble */
#endif
#ifndef _MEM_PARITY_WAR
#define      SEXC_ECC_EXCEPTION EXC_CODE(40) /* ECC/Parity error recovery */
#endif /* _MEM_PARITY_WAR */
#define      SEXC_UTINTR  EXC_CODE(41)     /* post-interrupt uthread processing */

/*
 * Coprocessor 0 operations
 */
#define      C0_READI    0x1              /* read ITLB entry addressed by C0_INDEX */
#define      C0_WRITEI   0x2              /* write ITLB entry addressed by C0_INDEX */
#define      C0_WRITER   0x6              /* write ITLB entry addressed by C0_RAND */
#define      C0_PROBE    0x8              /* probe for ITLB entry addressed by TLBHI */
#define      C0_RFE      0x10             /* restore for exception */
#define      C0_WAIT     0x20             /* wait for interrupt */

#if R4000

```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.k

```

/*
 * 'cache' instruction definitions
 */

/* Target cache */
#define      CACH_PI      0x0            /* specifies primary inst. cache */
#define      CACH_PD      0x1            /* primary data cache */
#define      CACH_SI      0x2            /* secondary instruction cache */
#define      CACH_SD      0x3            /* secondary data cache */

/* Cache operations */
#define      C_IINV       0x0            /* index invalidate (inst, 2nd inst) */
#define      C_IWBINV     0x0            /* index writeback inval (d, sd) */
#define      C_ILT        0x4            /* index load tag (all) */
#define      C_IST        0x8            /* index store tag (all) */
#define      C_CDX        0xc            /* create dirty exclusive (d, sd) */
#define      C_HINV       0x10           /* hit invalidate (all) */
#define      C_HWBINV     0x14           /* hit writeback inv. (d, sd) */
#define      C_FILL       0x14           /* fill (i) */
#define      C_HWB        0x18           /* hit writeback (i, d, sd) */
#define      C_HSV        0x1c           /* hit set virt. (si, sd) */
#ifndef TRITON
#define      C_INVALL     0x0            /* Triton invalidate all (s) */
#define      C_INVPAGE    0x14          /* Triton invalidate page (s) */
#endif /* TRITON */

/*
 * C0_CONFIG register definitions
 */
#define      CONFIG_CM    0x80000000     /* 1 == Master-Checker enabled */
#define      CONFIG_EC    0x70000000     /* System Clock ratio */
#define      CONFIG_EP    0x0f000000     /* Transmit Data Pattern */
#define      CONFIG_SB    0x00c00000     /* Secondary cache block size */

#define      CONFIG_SS    0x00200000     /* Split scache: 0 == I&D combined */
#define      CONFIG_SW    0x00100000     /* scache port: 0==128, 1==64 */
#define      CONFIG_EW    0x000c0000     /* System Port width: 0==64, 1==32 */
#define      CONFIG_SC    0x00020000     /* 0 -> 2nd cache present */
#define      CONFIG_SM    0x00010000     /* 0 -> Dirty Shared Coherency enabled */
#define      CONFIG_BE    0x00008000     /* Endian-ness: 1 --> BE */
#define      CONFIG_EM    0x00004000     /* 1 -> ECC mode, 0 -> parity */
#define      CONFIG_EB    0x00002000     /* Block order: 1->sequent, 0->subblock */

#define      CONFIG_IC    0x00000e00     /* Primary Icache size */

```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.l

```

#define CONFIG_DC 0x000001c0 /* Primary Dcache size */
#define CONFIG_IB 0x00000020 /* Icache block size */
#define CONFIG_DB 0x00000010 /* Dcache block size */
#define CONFIG_CU 0x00000008 /* Update on Store-conditional */
#define CONFIG_K0 0x00000007 /* K0SEG Coherency algorithm */

#ifdef TRITON
#define CONFIG_TR_SS 0x00300000 /* Triton SS (2nd cache size) */
#define CONFIG_TR_SC CONFIG_SC /* Triton SC (2nd cache present) */
#define CONFIG_TR_SE 0x00001000 /* Triton SE (2nd cache enabled) (R/W) */
#endif /* TRITON */

#define CONFIG_UNCACHED 0x00000002 /* K0 is uncached */
#ifdef _RUN_UNCACHED
#define CONFIG_NONCOHRNT CONFIG_UNCACHED
#define CONFIG_COHRNT_EXLWR CONFIG_UNCACHED
#define CONFIG_COHRNT_EXL CONFIG_UNCACHED
#else
#define CONFIG_NONCOHRNT 0x00000003
#define CONFIG_COHRNT_EXLWR 0x00000005
#define CONFIG_COHRNT_EXL 0x00000004
#endif
#ifdef R10000
#define CONFIG_UNCACHED_ACC 0x00000007
#endif /* R10000 */
#define CONFIG_SB_SHFT 22 /* shift SB to bit position 0 */
#define CONFIG_IC_SHFT 9 /* shift IC to bit position 0 */
#define CONFIG_DC_SHFT 6 /* shift DC to bit position 0 */
#define CONFIG_BE_SHFT 15 /* shift BE to bit position 0 */
#define CONFIG_IB_SHFT 5 /* shift IB to bit position 0 */
#define CONFIG_DB_SHFT 4 /* shift DB to bit position 0 */
#ifdef TRITON
#define CONFIG_TR_SS_SHFT 20 /* shift TR_SS to bit position 0 */
#endif /* TRITON */

/*
 * C0_TAGLO definitions for setting/getting cache states and physaddr bits
 */
#define SADDRMASK 0xFFFFE000 /* 31..13 -> scache paddr bits 35..17 */
#define SVINDEXMASK 0x00000380 /* 9..7: prim virt index bits 14..12 */
#define SSTATEMASK 0x00001c00 /* bits 12..10 hold scache line state */
#define SINVALID 0x00000000 /* invalid --> 000 == state 0 */
#define SCLEANEXCL 0x00001000 /* clean exclusive --> 100 == state 4 */

```

Appendix D-0.m

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

#define SDIRTYEXCL 0x00001400 /* dirty exclusive --> 101 == state 5 */
#define SECC_MASK 0x0000007f /* low 7 bits are ecc for the tag */
#define SADDR_SHIFT 4 /* shift STagLo (31..13) to 35..17 */

#define PADDRMASK 0xFFFFF00 /* PTagLo31..8->prim paddr bits35..12 */
#define PADDR_SHIFT 4 /* roll bits 35..12 down to 31..8 */
#define PSTATEMASK 0x00C0 /* bits 7..6 hold primary line state */
#define PINVALID 0x0000 /* invalid --> 000 == state 0 */
#define PCLEANEXCL 0x0080 /* clean exclusive --> 10 == state 2 */
#define PDIRTYEXCL 0x00C0 /* dirty exclusive --> 11 == state 3 */
#define PPARITY_MASK 0x0001 /* low bit is parity bit (even). */

/*
 * C0_CACHE_ERR definitions.
 */
#define CACHERR_ER 0x80000000 /* 0: inst ref, 1: data ref */
#define CACHERR_EC 0x40000000 /* 0: primary, 1: secondary */
#define CACHERR_ED 0x20000000 /* 1: data error */
#define CACHERR_ET 0x10000000 /* 1: tag error */
#define CACHERR_ES 0x08000000 /* 1: external ref, e.g. snoop */
#define CACHERR_EE 0x04000000 /* error on SysAD bus */
#define CACHERR_EB 0x02000000 /* complicated, see spec. */
#define CACHERR_EI 0x01000000 /* complicated, see spec. */
#ifdef IPL9
#define CACHERR_EW 0x00800000 /* complicated, see spec. */
#endif
#define CACHERR_SIDX_MASK 0x003ffff8 /* secondary cache index */
#define CACHERR_PIDX_MASK 0x00000007 /* primary cache index */
#define CACHERR_PIDX_SHIFT 12 /* bits 2..0 are paddr14..12 */

#endif /* R4000 */

#ifdef R4000 || R10000
/* The R4000 and R10000 families supports hardware watchpoints:
 * C0_WATCHLO:
 * bits 31..3 are bits 31..3 of physaddr to watch
 * bit 2: reserved; must be written as 0.
 * bit 1: when set causes a watchpoint trap on load accesses to paddr.
 * bit 0: when set traps on stores to paddr;
 * C0_WATCHHI
 * bits 31..4 are reserved and must be written as zeros - R4000
 * bits 3..0 are bits 35..32 of the physaddr to watch - R4000
 * bits 31..8 are reserved and must be written as zeros - R10000
 * bits 3..0 are bits 39..32 of the physaddr to watch - R10000

```

Appendix D-0.n

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

*/
#define WATCHLO_WTRAP          0x00000001
#define WATCHLO_RTRAP         0x00000002
#define WATCHLO_ADDRMASK      0xffffffff8
#define WATCHLO_VALIDMASK     0xffffffffb
#if R4000 && (! defined(_NO_R4000))
#define WATCHHI_VALIDMASK     0x0000000f
#else R10000
#define WATCHHI_VALIDMASK     0x000000ff
#endif
#endif /* R4000 || R10000 */

/*
 * Coprocessor 0 registers
 * Some of these are r4000 specific.
 */
#ifdef _LANGUAGE_ASSEMBLY
#define CO_INX                  $0
#define CO_RAND                 $1
#define CO_TLBLO                $2
#define CO_CTXT                 $4
#define CO_BADVADDR             $8
#define CO_TLBHI                $10
#define CO_SR                   $12
#define CO_CAUSE                 $13
#define CO_EPC                  $14
#define CO_PRID                 $15          /* revision identifier */

#if R4000 || R10000
#define CO_TLBLO_0              $2
#define CO_TLBLO_1              $3
#define CO_PGMASK               $5          /* page mask */
#define CO_TLBWired             $6          /* # wired entries in tlb */
#define CO_COUNT                $9          /* free-running counter */
#define CO_COMPARE              $11         /* counter comparison reg. */
#define CO_CONFIG               $16         /* hardware configuration */
#define CO_LLADDR               $17         /* load linked address */
#define CO_WATCHLO              $18         /* watchpoint */
#define CO_WATCHHI              $19         /* watchpoint */
#define CO_EXTCTXT              $20         /* Extended context */
#define CO_ECC                  $26         /* S-cache ECC and primary parity */
#define CO_CACHE_ERR            $27         /* cache error status */
#define CO_TAGLO                $28         /* cache operations */
#endif

#endif


```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.o

```

#define CO_TAGHI                 $29          /* cache operations */
#define CO_ERROR_EPC            $30          /* ECC error prg. counter */
#endif /* R4000 || R10000 */

#ifdef R10000
#define CO_FMMASK               $21          /* Frame Mask */
#define CO_BRDIAG               $22         /* Indices of tlb wired entries */
#define CO_PRFCNT0              $25         /* performance counter 0 */
#define CO_PRFCNT1              $25         /* performance counter 1 */
#define CO_PRFCRTL0             $25         /* performance control reg 0 */
#define CO_PRFCRTL1             $25         /* performance control reg 1 */
#endif /* R10000 */

# else /* ! _LANGUAGE_ASSEMBLY */
#define CO_INX                  0
#define CO_RAND                 1
#define CO_TLBLO                2
#define CO_CTXT                 4
#define CO_BADVADDR             8
#define CO_TLBHI                10
#define CO_SR                   12
#define CO_CAUSE                 13
#define CO_EPC                  14
#define CO_PRID                 15          /* revision identifier */

#if R4000 || R10000
#define CO_TLBLO_0              2
#define CO_TLBLO_1              3
#define CO_PGMASK               5          /* page mask */
#define CO_TLBWired             6          /* # wired entries in tlb */
#define CO_COUNT                9          /* free-running counter */
#define CO_COMPARE              11         /* counter comparison reg. */
#define CO_CONFIG               16         /* hardware configuration */
#define CO_LLADDR               17         /* load linked address */
#define CO_WATCHLO              18         /* watchpoint */
#define CO_WATCHHI              19         /* watchpoint */
#define CO_EXTCTXT              20         /* Extended context */
#define CO_ECC                  26         /* S-cache ECC and primary parity */
#define CO_CACHE_ERR            27         /* cache error status */
#define CO_TAGLO                28         /* cache operations */
#define CO_TAGHI                29         /* cache operations */
#define CO_ERROR_EPC            30         /* ECC error prg. counter */
#endif

#endif


```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix D-0.p

```

#ifdef R10000
#define CO_FMMASK 21 /* Frame Mask */
#define CO_BRDIAG 22 /* Indices of tlb wired entries */
#define CO_PRCNT0 25 /* performance counter 0 */
#define CO_PRCNT1 25 /* performance counter 1 */
#define CO_PRCRTL0 25 /* performance control reg 0 */
#define CO_PRCRTL1 25 /* performance control reg 1 */
#endif /* R10000 */

#endif /* _LANGUAGE_ASSEMBLY */

#ifdef R10000
#include "sys/R10k.h"
#endif /* R10000 */

#endif /* R4000 || R10000 */

#if _MIPS_SIM == _ABIO32
#define _S_EXT_(addr) (addr)
#else
#define _S_EXT_(addr) ((addr) | 0xffffffff00000000)
#endif

/* CO_PRID Defines common to all cpus' */
/*
 * coprocessor revision identifiers
 */
#ifdef _KERNEL
#ifdef _LANGUAGE_C
typedef union rev_id {
    unsigned int ri_uint;
    struct {
#ifdef MIPSEB
        unsigned int Ri_fill:16,
                    Ri_imp:8, /* implementation id */
                    Ri_majrev:4, /* major revision */
                    Ri_minrev:4; /* minor revision */
#else
        unsigned int Ri_minrev:4, /* minor revision */
                    Ri_majrev:4, /* major revision */
                    Ri_imp:8, /* implementation id */
                    Ri_fill:16;
#endif
    };
} rev_id_t;
#endif /* _LANGUAGE_C */
#endif /* _KERNEL */

#endif /* MIPSEL */

```

Appendix D-0.q

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```

#endif /* MIPSEL */
} Ri;
} rev_id_t;
#define ri_imp Ri.Ri_imp
#define ri_majrev Ri.Ri_majrev
#define ri_minrev Ri.Ri_minrev
#endif /* _LANGUAGE_C */
#endif /* _KERNEL */

#define CO_IMP_MASK 0xff00
#define CO_IMP_SHIFT 8
#define CO_REV_MASK 0xff
#define CO_MAJREV_MASK 0xf0
#define CO_MAJREV_SHIFT 4
#define CO_MINREV_MASK 0xf
#define CO_MINREV_SHIFT 0

#define CO_IMP_UNDEFINED 0x24
#define CO_IMP_R5000 0x23
#define CO_IMP_TRITON CO_IMP_R5000
#define CO_IMP_R4650 0x22
#define CO_IMP_R4700 0x21
#define CO_IMP_R4600 0x20
#define CO_IMP_R8000 0x10
#define CO_IMP_R12000 0x0e
#define CO_IMP_R10000 0x09
#define CO_IMP_R6000A 0x06
#define CO_IMP_R4400 0x04
#define CO_MAJREVMIN_R4400 0x04
#define CO_IMP_R4000 0x04
#define CO_IMP_R6000 0x03
#define CO_IMP_R3000A 0x02
#define CO_MAJREVMIN_R3000A 0x03
#define CO_IMP_R3000 0x02
#define CO_MAJREVMIN_R3000 0x02
#define CO_IMP_R2000A 0x02
#define CO_MAJREVMIN_R2000A 0x01
#define CO_IMP_R2000 0x01

#define CO_MAKE_REVID(x,y,z) (((x) << CO_IMP_SHIFT) | \
                              ((y) << CO_MAJREV_SHIFT) | \
                              ((z) << CO_MINREV_SHIFT))

/*
 * Defines for the CO_PGMASK register.
 */

```

Appendix D-0.r

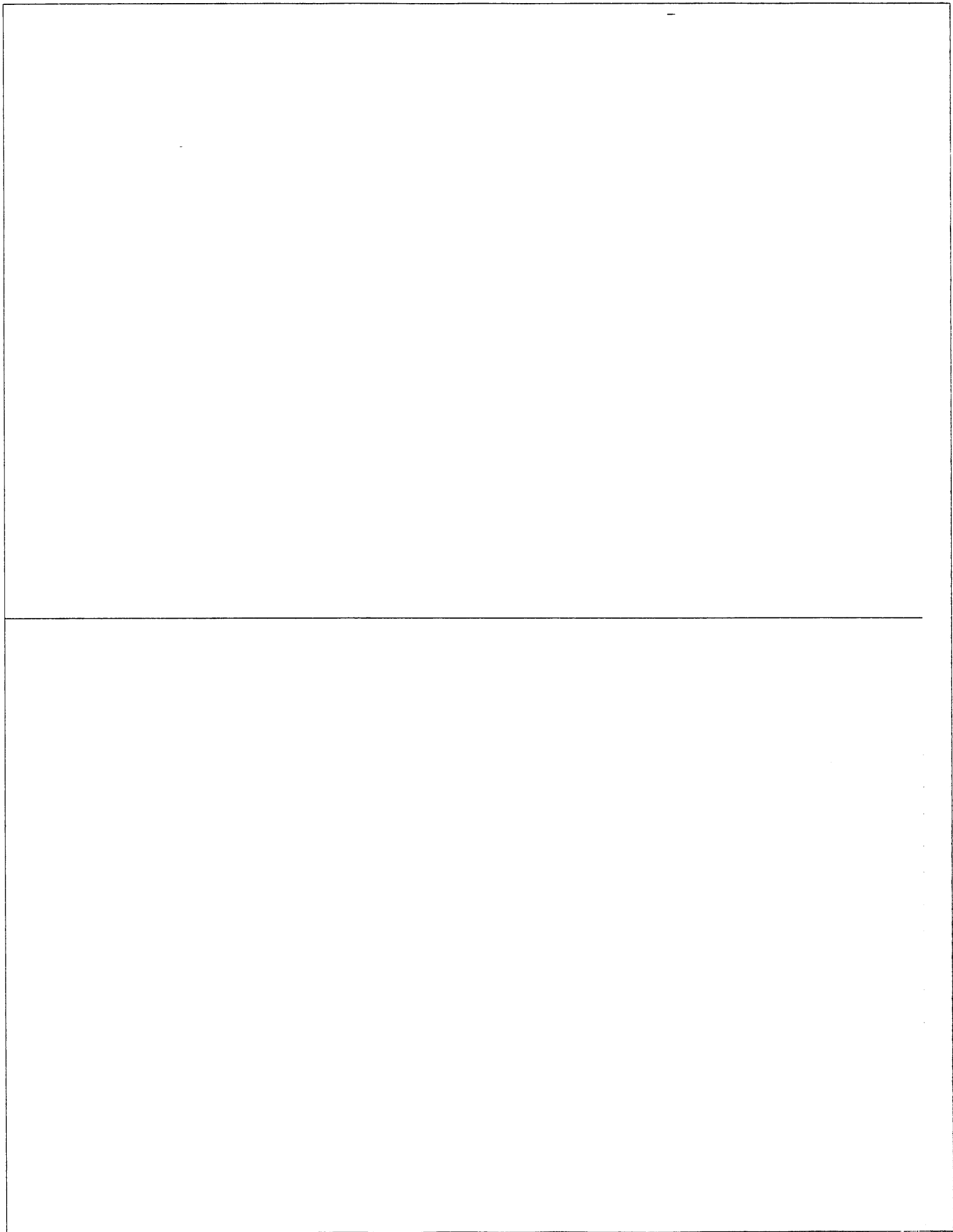
22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
*/
#define defined (R10000) || defined (R4000)
#define PGMASK_SHFT 13
#endif

#ifdef _KERNEL
#ifdef _LANGUAGE_C
#ifdef R10000
#ifdef R4000
extern int get_cpu_irr(void);
#define IS_R10000() ((get_cpu_irr() >> CO_IMPSHIFT) == CO_IMP_R10000)
#else /* R4000 */
#define IS_R10000() (1)
#endif /* R4000 */
#else /* R10000 */
#define IS_R10000() (0)
#endif /* R10000 */
#endif /* _LANGUAGE_C */
#endif /* _KERNEL */

#endif /* __SYS_SBD_H__ */
```



Appendix E: kldir.h header file - has map of kernel in low memory

kldir.h header file (excerpt)

```

=====
kldir.h header file
/hosts/bonnie.engr.sgi.com/proj/irix6.5se/isms/irix/kern/sys/kldir.h
=====
/*****
 *
 *      Copyright (C) 1992-1997, Silicon Graphics, Inc.
 *
 *  These coded instructions, statements, and computer programs contain
 *  unpublished proprietary information of Silicon Graphics, Inc., and
 *  are protected by Federal copyright law. They may not be disclosed
 *  to third parties or copied or duplicated in any form, in whole or
 *  in part, without the prior written consent of Silicon Graphics, Inc.
 *
 *****/

#ifndef __SYS_SN_KLDIR_H__
#define __SYS_SN_KLDIR_H__

#define "$Revision: 1.20 $"

/*
 * The kldir memory area resides at a fixed place in each node's memory and
 * provides pointers to most other SNO memory areas. This allows us to
 * resize and/or relocate memory areas at a later time without breaking all
 * firmware and kernels that use them. Indices in the array are
 * permanently dedicated to areas listed below. Some memory areas (marked
 * below) reside at a permanently fixed location, but are included in the
 * directory for completeness.
 */

#define KLDIR_MAGIC          0x434d5f53505f5357

/*
 * The upper portion of the memory map applies during boot
 * only and is overwritten by IRIX/SYMMON.

```

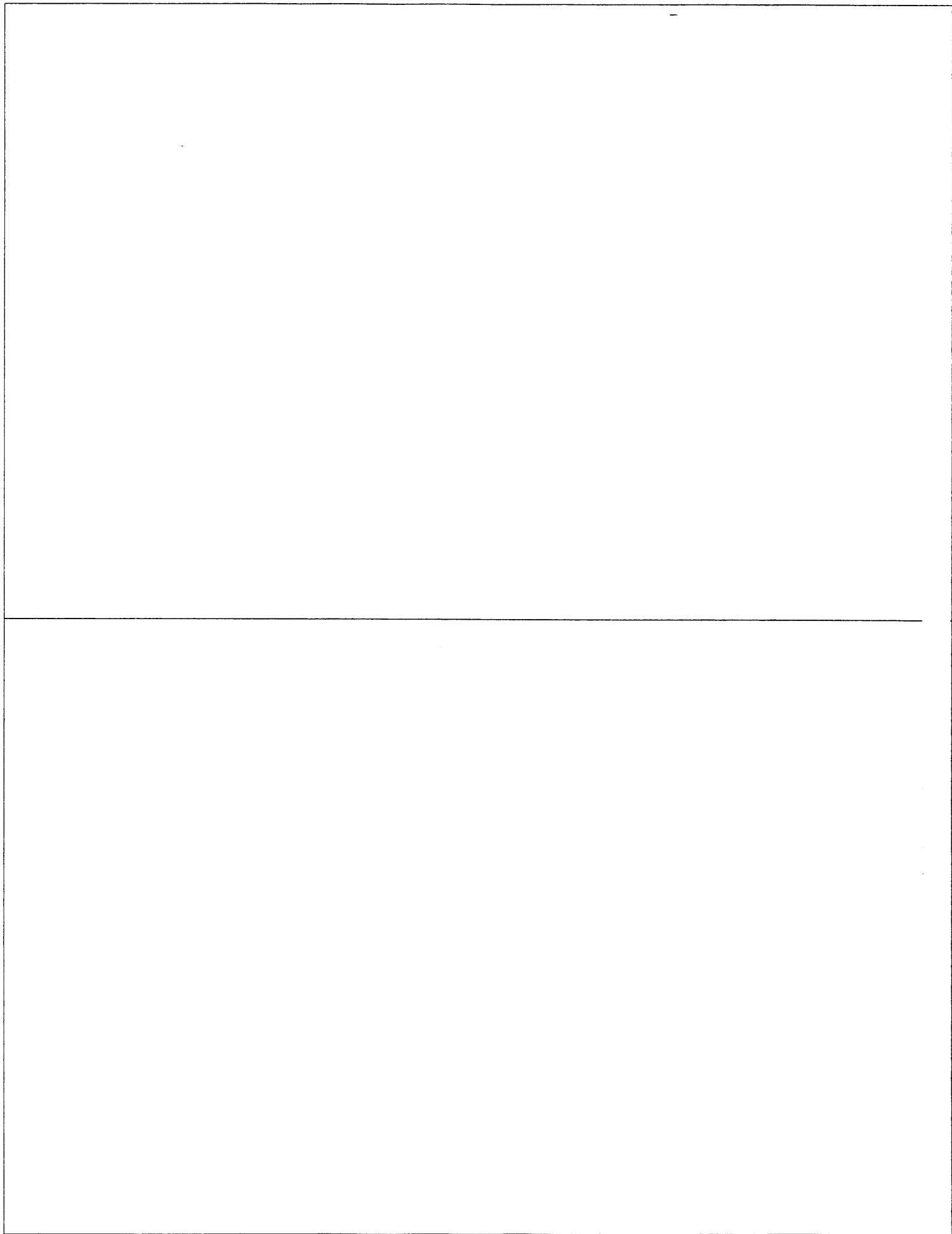
```

 *
 *      MEMORY MAP PER NODE
 *
 * 0x2000000 (32M)  |-----|
 *                  | IO6 BUFFERS FOR FLASH ENET IOC3 |
 * 0x1F80000 (31.5M) |-----|
 *                  | IO6 TEXT/DATA/BSS/stack |
 * 0x1C00000 (30M)  |-----|
 *                  | IO6 PROM DEBUG TEXT/DATA/BSS/stack |
 * 0x0800000 (28M)  |-----|
 *                  | IP27 PROM TEXT/DATA/BSS/stack |
 * 0x1B00000 (27M)  |-----|
 *                  | IP27 CFG |
 * 0x1A00000 (26M)  |-----|
 *                  | Graphics PROM |
 * 0x1800000 (24M)  |-----|
 *                  | 3rd Party PROM drivers |
 * 0x1600000 (22M)  |-----|
 *                  | Free |
 *                  |-----|
 *                  | UNIX DEBUG Version |
 *                  | SYMMON |
 *                  | (For UNIX Debug only) |
 * 0x34000 (208K)   |-----|
 *                  | SYMMON STACK [NUM_CPU_PER_NODE] |
 *                  | (For UNIX Debug only) |
 * 0x25000 (148K)   |-----|
 *                  | KLCONFIG - II (temp) |
 *                  |-----|
 *                  | UNIX NON-DEBUG Version |
 * 0x19000 (100K)   |-----|
 *
 * The lower portion of the memory map contains information that is
 * permanent and is used by the IP27PROM, IO6PROM and IRIX.
 *
 * 0x19000 (100K)   |-----|
 *                  | PI Error Spools (32K) |
 *

```

* 0x12000 (72K)	Unused
* 0x11c00 (71K)	CPU 1 NMI Eframe area
* 0x11a00 (70.5K)	CPU 0 NMI Eframe area
* 0x11800 (70K)	CPU 1 NMI Register save area
* 0x11600 (69.5K)	CPU 0 NMI Register save area
* 0x11400 (69K)	GDA (1k)
* 0x11000 (68K)	Early cache Exception stack and/or kernel/io6prom nmi registers
* 0x10800 (66k)	cache error eframe
* 0x10400 (65K)	Exception Handlers (UALIAS copy)
* 0x10000 (64K)	
* *	KLCONFIG - I (permanent) (48K)
* *	
* 0x4000 (16K)	NMI Handler (Protected Page)
* 0x3000 (12K)	ARCS PVECTORS (master node only)
* 0x2c00 (11K)	ARCS TVECTORS (master node only)
* 0x2800 (10K)	LAUNCH [NUM_CPU]
* 0x2400 (9K)	Low memory directory (KLDIR)
* 0x2000 (8K)	ARCS SPB (1K)
* 0x1000 (4K)	Early cache Exception stack and/or kernel/io6prom nmi registers
* *	

* 0x800 (2k)	cache error eframe
* 0x400 (1K)	Exception Handlers
* 0x0 (0K)	



Appendix F: IRIX 6.5 Kernel Values

IRIX 6.5 Kernel Values

Unit covers:

- List of kernel system values and constants extracted by icrash
- Examples of 32 and 64 bit (IRIX 6.5) systems
- Short definition of values
- Sample output of training icrash "kerninfo" command

Kernel Value Table

The following table displays key constant and define values for the IRIX 6.5 kernel (Beta).
Please consider this table work-in-progress at the moment, other information will be added at a later date.

Column Meanings

icrash Name

Name of value in icrash.

NOTE: assume the prefix "K_" before each name if looking for actual name in icrash.

icrash Structure

Name of structure in icrash where value defined/stored.

Indy Live System

Sample values of an Indy Workstation running IRIX 6.5 (Beta).

O2000 dump

Sample values from an O2000 dump of IRIX 6.5 beta (called IRIX64).

O2000 live system

Sample values from recent (AriI 1998) O2000 (flurry) running IRIX 6.5 beta (called IRIX64).

Description

Short description of field meaning/usage.

Actual kernel name equivalents specified as "kernel: *name*".

Kernel Value Table

icrash Name K_XXXXXXXXXX	icrash Structure Name	Indy Live System (32 bit machine)	O2000 Dump (64 bit machine)	O2000 Live System (64 bit machine)	Description
ACTIVEFILES	global_s	0x0 (0)	0x0 (0)	0x0 (0)	Kernel: activefiles linked list of active file (DEBUG only)
BLOCK_ALLOC	callback_s	0x10014b40 (268520256)	0x10014b70 (268520304)	0x10014b70 (268520304)	
BLOCK_FREE	callback_s	0x10014ba8 (268520360)	0x10014be0 (268520416)	0x10014be0 (268520416)	
CORE TYPE	coreinfo_s	/dev/kmem	corefile	/dev/kmem	Core type: dump="corefile" live sys="/dev/kmem"
COREFILE	coreinfo_s	/dev/mem	vmcore.15.comp	/dev/mem	Core (file) name
CORE_FD	coreinfo_s	0x3 (3)	0x3 (3)	0x4 (4)	
DEFKTHREAD	global_s	0x0 (0)	0xa800000103897000	0x0 (0)	"Current " default kernel thread
DUMPCPU	global_s	0x0 (0)	0x4 (4)	0x0 (0)	CPU that executed dumpsys()
DUMPKTHREAD	global_s	0x0 (0)	0xa800000103897000	0x0 (0)	Kernel thread that executed dumpsys()
DUMPPROC	global_s	0x0 (0)	0xa80000010261b000	0x0 (0)	If dump, pointer to proc that executed dumpsys()
DUMPREGS	global_s	0x0 (0)	0x10e627d8 (283518936)	0x0 (0)	Kernel:dumpregs: Area where PANIC registers are saved
DUMP_HDR	coreinfo_s	(null)	CrshDump	(null)	Header from dump corefile
END	sysinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	
ERROR_DUMPBUF	global_s	0x0 (0)	0x0 (0)	0x0 (0)	Kernel:error_dumpbuf EVEREST only
EXTSTKIDX	kerninfo_s	0x1 (1)	0x0 (0)	0x0 (0)	Extend stack (yes/no) 32 bit kernels only
EXTUSIZE	kerninfo_s	0x1 (1)	0x0 (0)	0x0 (0)	Extend stack index 32 bit kernels only
FLAGS	global_s	0x0 (0)	0x0 (0)	0x0 (0)	

HWGRAPH	global_s	0x10ab1000 (279646208)	0x10e77000 (283602944)	0x10dff000 (283111424)	
HWGRAPHP	global_s	0xc0002000	0xa80000000055a000	0xa800000001370000	
ICRASHDEF	coreinfo_s	(null)	(null)	(null)	icrash definition file if sepcified on comand line
IP	sysinfo_s	0x16 (22)	0x1b (27)	0x1b (27)	Processor IPxx
IRIX_REV	kerninfo_s	0x5 (5)	0x5 (5)	0x5 (5)	IRIX revision: IRIX6_x
KOBASE	kerninfo_s	0x80000000	0xa800000000000000	0xa800000000000000	Base of kernel K0 memory
KOSIZE	kerninfo_s	0x20000000 (536870912)	0x1000000000	0x1000000000	Size of kernel K0 memory
K1BASE	kerninfo_s	0xa0000000	0x9600000000000000	0x9600000000000000	Base of kernel K1 memory
K1SIZE	kerninfo_s	0x20000000 (536870912)	0x1000000000	0x1000000000	Size of kernel K1 memory
K2BASE	kerninfo_s	0xc0000000	0xc000000000000000	0xc000000000000000	Base of kernel K2 memory
K2SIZE	kerninfo_s	0x20000000 (536870912)	0xffff80000000	0xffff80000000	Size of kernel K2 memory
KERNELSTACK	kerninfo_s	0xffffc000	0xffffffffffff8000	0xffffffffffff8000	End of stack "page" start here for stack traces
KERNSTACK	kerninfo_s	0xffffd000	0xffffffffffffc000	0xffffffffffffc000	Base of kernel stack area (mapped on all CPUs)
KEXTSTACK	kerninfo_s	0xffffb000	0x0 (0)	0x0 (0)	Extend stack amount 32 bit kernels only
KPTBL	global_s	0x8838c000	0xa800000000540000	0xa80000000071c000	Kemel:kptbl Base of kernel page (PDE) table
KPTBLP	global_s	0x0 (0)	0x10e79008 (283611144)	0x0 (0)	icrash (local) copy of kptbl (dump only)
KPTEBASE	kerninfo_s	0xff800000	0xc0000fc000000000	0xc0000fc000000000	Base of segment table for regular and sprroc processes (threads)
KPTE_SHDUBASE	kerninfo_s	0xffffffffffff800000	0xc00007c000000000	0xc00007c000000000	Base of user segment (pde) table
KPTE_USIZE	kerninfo_s	0x200000000	0x8000000000	0x8000000000	Size of segment (pde) table for user processes
KSTKIDX	kerninfo_s	0x0 (0)	0x0 (0)	0x0 (0)	
KUBASE	kerninfo_s	0x0 (0)	0x0 (0)	0x0 (0)	Lower limit of user address space
KUSIZE	kerninfo_s	0x80000000	0x1000000000	0x1000000000	Upper limit of user address space (size)

Appendix F-4.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

LBOLT	global_s	0x88337af4	0xc00000000144d284	0xc00000000145d80c	Kemel:lbolt time in HZ since last boot
MAPPED_PAGE_SIZE	kerninfo_s	0x0 (0)	0x1000000 (16777216)	0x1000000 (16777216)	Size of mapped kernel's mapped page size
MAPPED_RO_BASE	kerninfo_s	0x0 (0)	0xc000000000000000	0xc000000000000000	Kemel K2 memory base (K2BASE), read-only portion
MAPPED_RW_BASE	kerninfo_s	0x0 (0)	0xc000000001000000	0xc000000001000000	Kemel K2 memory base of read-write portion
MASTER_NASID	sysinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	Node ID of "master" node
MAXCPUS	sysinfo_s	0x1 (1)	0x6 (6)	0x80 (128)	Number of CPUs on the system
MAXNODES	sysinfo_s	0x1 (1)	0x3 (3)	0x40 (64)	Number of nodes on the system
MAXPFN	kerninfo_s	0x1ffff (131071)	0x3ffffff (67108863)	0x3ffffff (67108863)	Highest PFN (page frame number) on the system (physmem>>pnumshift)
MAXPHYS	kerninfo_s	0x1ffffff (536870911)	0xffffffff	0xffffffff	Maximum physical (portion of) address same as K_TO_PHYS_MASK
MEMBER_BASEVAL	callback_s	0x10014ab0 (268520112)	0x10014ae0 (268520160)	0x10014ae0 (268520160)	
MEMBER_BITLEN	callback_s	0x10014a88 (268520072)	0x10014ab0 (268520112)	0x10014ab0 (268520112)	
MEMBER_OFFSET	callback_s	0x10014980 (268519808)	0x100149a0 (268519840)	0x100149a0 (268519840)	
MEMBER_SIZE	callback_s	0x10014a00 (268519936)	0x10014a20 (268519968)	0x10014a20 (268519968)	
MEM_PER_BANK	sysinfo_s	0x0 (0)	0x20000000 (536870912)	0x20000000 (536870912)	Memory per node divided by banks per node (4 or 8)
MEM_PER_NODE	sysinfo_s	0x0 (0)	0x10000000	0x10000000	(1 << NASID_SHIFT (below))
MEM_PER_SLOT	sysinfo_s	0x0 (0)	0x8000000 (134217728)	0x8000000 (134217728)	
MLINFOLIST	global_s	0x885dc2a0	0xa80000010084efa0	0xa8000008009240c0	
NAMELIST	coreinfo_s	/unix	/unix.15	/unix	
NASID_BITMASK	sysinfo_s	0x0 (0)	0xff (255)	0xff (255)	
NASID_SHIFT	sysinfo_s	0x0 (0)	0x20 (32)	0x20 (32)	Node ID (NASID) shift amount position of NASID in addr
NBPC	kerninfo_s	0x1000 (4096)	0x4000 (16384)	0x4000 (16384)	Number bytes per "click" (_PAGESZ)

Appendix F-4.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

NBPS	kerninfo_s	0x400000 (4194304)	0x2000000 (33554432)	0x2000000 (33554432)	Number bytes per segment (Bytes.per.page * pages.per.segment)
NBPW	kerninfo_s	0x4 (4)	0x8 (8)	0x8 (8)	Number bytes per word (32/8) or (64/8)
NCPS	kerninfo_s	0x400 (1024)	0x800 (2048)	0x800 (2048)	Number clicks per segment Bytes.per.click / PDE size(8))
NODEPDAINDR	global_s	0x88338010	0xc000000014cf9d0	0xc000000014bd288	Kernel:Nodepdaindr Array of addresses to node private data areass
NPROCS	kerninfo_s	0x1a8 (424)	0x528 (1320)	0x5688 (22152)	Kernel:v_proc Maximum number of user processes
NTLBENTRIES	sysinfo_s	0x0 (0)	0x40 (64)	0x0 (0)	CPU TLB size based on IP number dump only
NUMCPUS	sysinfo_s	0x1 (1)	0x6 (6)	0x80 (128)	Number CPUs in system
NUMNODES	sysinfo_s	0x1 (1)	0x3 (3)	0x40 (64)	Number nodes in system
PAGESZ	kerninfo_s	0x1000 (4096)	0x4000 (16384)	0x4000 (16384)	Working pages size in system (_ PAGESZ)
PANIC_TYPE	coreinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	Panic type "1" means NMI
PARCELS_PER_SLOT	sysinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	
PARCEL_BITMASK	sysinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	
PARCEL_SHIFT	sysinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	
PDAINDR	global_s	0x88337488	0xc0000000144e5a0	0xc0000000145eaa0	Kernel: pdaindr Array pointing to CPU PDA (private data area)
PDE_PG_CC	pdeinfo_s	0x38 (56)	0x38 (56)	0x38 (56)	PDE CC bit mask
PDE_PG_D	pdeinfo_s	0x80000000	0x200000000	0x200000000	PDE D bit mask
PDE_PG_EOP	pdeinfo_s	0x4000000 (67108864)	0x0 (0)	0x0 (0)	PDE EndOfPage bit mask
PDE_PG_G	pdeinfo_s	0x1 (1)	0x1 (1)	0x1 (1)	PDE G bit mask
PDE_PG_M	pdeinfo_s	0x4 (4)	0x4 (4)	0x4 (4)	PDE M bit mask
PDE_PG_N	pdeinfo_s	0x10 (16)	0x28 (40)	0x28 (40)	PDE N bit mask
PDE_PG_NR	pdeinfo_s	0x1800000 (402653184)	0x400000000	0x400000000	PDE NR bit mask

PDE_PG_SV	pdeinfo_s	0x4000000 (1073741824)	0x100000000	0x100000000	PDE SV bit mask
PDE_PG_VR	pdeinfo_s	0x2 (2)	0x2 (2)	0x2 (2)	PDE VR bit mask
PFDAT	global_s	0x8827c128	0x0 (0)	0x0 (0)	Kernel: pfdat non-NUMA: pfdat table address NUMA: see Kernel p_nodepda in PDA for node resident pdat addresses
PFN_MASK	pdeinfo_s	0x3ffffc0 (67108800)	0xffffffff00	0xffffffff00	Mask to extract PFN from address
PFN_SHIFT	pdeinfo_s	0x6 (6)	0x8 (8)	0x8 (8)	PFN shift amount, rightmost bit in address
PG_CC_SHIFT	pdeinfo_s	0x3 (3)	0x3 (3)	0x3 (3)	PDE CC shift amount
PG_D_SHIFT	pdeinfo_s	0x1f (31)	0x21 (33)	0x21 (33)	PDE D shift amount
PG_EOP_SHIFT	pdeinfo_s	0x1a (26)	0xffffffffffffff	0xffffffffffffff	PDE EndOfPage bit shift amount
PG_G_SHIFT	pdeinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	PDE G bit shift amount
PG_M_SHIFT	pdeinfo_s	0x2 (2)	0x2 (2)	0x2 (2)	PDE M bit shift amount
PG_NR_SHIFT	pdeinfo_s	0x1b (27)	0x22 (34)	0x22 (34)	PDE NR bit shift amount
PG_N_SHIFT	pdeinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	PDE N bit shift amount
PG_SV_SHIFT	pdeinfo_s	0x1e (30)	0x20 (32)	0x20 (32)	PDE SV bit shift amount
PG_VR_SHIFT	pdeinfo_s	0x1 (1)	0x1 (1)	0x1 (1)	PDE VR bit shift amount
PHYSMEM	sysinfo_s	0x6000 (24576)	0x5000 (20480)	0x23c000 (2342912)	Kernel: physmem Physical memory size in pages
PIDACTIVE	global_s	0x88338090	0xc000000014d0788	0xc000000014be040	Kernel: pidactive list of active pids (processes)
PIDTAB	global_s	0xc0026000	0xa80000000f14000	0xc000000003ab4000	Kernel: pidtab pid table array address
PIDTABSZ	global_s	0x1a8 (424)	0x528 (1320)	0x5688 (22152)	Kernel: pidtabsz (v_proc) size of pidtab
PID_BASE	global_s	0x0 (0)	0x0 (0)	0x0 (0)	Kernel: pid_base CELL only
PNUMSHIFT	kerninfo_s	0xc (12)	0xe (14)	0xe (14)	PNUM shift amount, right bit of PNUM in address
PRINT_ERROR	callback_s	0x1009c678 (269084280)	0x1009f760 (269088608)	0x1009f760 (269088608)	
PROGRAM	global_s	icrash	icrash/icrash	icrash	icrash called as ____ (or fru)

PTRSZ	kerninfo_s	0x20 (32)	0x40 (64)	0x40 (64)	Size of pointer in bits used to determine word size of kernel
PUTBUF	global_s	0x8839bc00	0xa8000000051b000	0xa800000005c2c00	Kernel: putbuf address of kernel put (console message) buffer
RAM_OFFSET	kerninfo_s	0x8000000 (134217728)	0x0 (0)	0x0 (0)	Kernel: _physmem_start
REGSZ	kerninfo_s	0x8 (8)	0x8 (8)	0x8 (8)	Register size in bytes (always 8)
RW_FLAG	coreinfo_s	0x2 (2)	0x0 (0)	0x2 (2)	icrash core file read-write flag core dump always read-only
SLOTS_PER_NODE	sysinfo_s	0x0 (0)	0x20 (32)	0x20 (32)	Kernel: slots_per_node lots per node (SN only)
SLOT_BITMASK	sysinfo_s	0x0 (0)	0x1f (31)	0x1f (31)	Kernel: slot_bitmask extract slot number from address (SN only)
SLOT_SHIFT	sysinfo_s	0x0 (0)	0x1b (27)	0x1b (27)	Kernel: slot_shift slot shift amount, right bit of slot number in address(SN only)
STHREADLIST	global_s	0x8835e710	0xc0000000014cd370	0xc0000000014bab08	Kernel: sthreadlist service thread linked list
STRST	global_s	0x0 (0)	0x0 (0)	0x0 (0)	Kernel: strst STREAMS statistics structure
STRUCT_LEN	callback_s	0x10014908 (268519688)	0x10014920 (268519712)	0x10014920 (268519712)	
SYM_ADDR	callback_s	0x10014870 (268519536)	0x10014880 (268519552)	0x10014880 (268519552)	
SYSTEMSIZE	sysinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	? (unset)
SYSSEGSZ	kerninfo_s	0x3800 (14336)	0x2800 (10240)	0x11e000 (1171456)	Kernel: syssegsz tuning variable "syssegsz" specifies the size of the K2 (dynamically allocated) memory pool
SYSTEMSIZE	sysinfo_s	0x0 (0)	0x0 (0)	0x0 (0)	? (unset)
TIME	global_s	0x88337af0	0xc00000000144d280	0xc00000000145d808	Kernel: time time in seconds since 1970
TLBDUMPSIZE	sysinfo_s	0x0 (0)	0x608 (1544)	0x0 (0)	Size of TLB table in dump (dump only)

Appendix F-4.e

22jul1998

TR-IKI rev 0.7b SGI Proprietary

TLBENTRYSZ	sysinfo_s	0x0 (0)	0x18 (24)	0x0 (0)	Size of a TLB entry in the dump (dump only)
TO_PHYS_MASK	kerninfo_s	0x1fffffff (536870911)	0xffffffff	0xffffffff	Mask to extract physical part of address
UPGIDX	kerninfo_s	0xffffffffffffff	0xffffffffffffff	0xffffffffffffff	-1 (not used)
USIZE	kerninfo_s	0x1 (1)	0x1 (1)	0x1 (1)	Number of pages in the kernel stack
UTSNAME	sysinfo_s	IRIX	IRIX64	IRIX64	Kernel: utsname from name.c
XTHREADLIST	global_s	0x883575d8	0xc000000001482a70	0xc0000000014b6060	Kernel: xthreadlist Kernel interrupt thread list

Appendix F-4.f

22jul1998

TR-IKI rev 0.7b SGI Proprietary

Sample "kerninfo" output

Code for the "kerninfo" command was added to icrash as an instrument of study in order to produce this table. The code for this command is not (yet) available in supported icrash, but will be made available from training upon request.

The following icrash output as used to create the table above.

Live Indy Workstation (IRIX 6.5 beta)

```
>> kerninfo
icrash coreinfo_s data fields
```

```
CORE_TYPE: /dev/kmem
PANIC_TYPE: 0x0 (0)
COREFILE: /dev/mem
NAMELIST: /unix
ICRASHDEF: (null)
CORE_FD: 0x3 (3)
DUMP_HDR: (null)
RW_FLAG: 0x2 (2)
```

```
icrash sysinfo_s data fields
```

```
UTSNAME: IRIX
IP: 0x16 (22)
PHYSMEM: 0x6000 (24576)
NUMCPUS: 0x1 (1)
MAXCPUS: 0x1 (1)
NTLBENTRIES: 0x0 (0)
TLBENTRYSZ: 0x0 (0)
TLBDUMPSIZE: 0x0 (0)
NUMNODES: 0x1 (1)
MAXNODES: 0x1 (1)
MASTER_NASID: 0x0 (0)
NASID_SHIFT: 0x0 (0)
SLOT_SHIFT: 0x0 (0)
PARCEL_SHIFT: 0x0 (0)
PARCEL_BITMASK: 0x0 (0)
PARCELS_PER_SLOT: 0x0 (0)
SLOTS_PER_NODE: 0x0 (0)
MEM_PER_NODE: 0x0 (0)
MEM_PER_SLOT: 0x0 (0)
MEM_PER_BANK: 0x0 (0)
NASID_BITMASK: 0x0 (0)
SLOT_BITMASK: 0x0 (0)
SYSTEMSIZE: 0x0 (0)
SYSTEMSIZE: 0x0 (0)
END: 0x0 (0)
```

```
icrash kerninfo_s data fields
```

IRIX_REV: 0x5 (5)
SYSSEGSZ: 0x3800 (14336)
NPROCS: 0x1a8 (424)
PTRSZ: 0x20 (32)
REGSZ: 0x8 (8)
PAGESZ: 0x1000 (4096)
NBPW: 0x4 (4)
NBPC: 0x1000 (4096)
NCPS: 0x400 (1024)
NBPS: 0x400000 (4194304)
PNUMSHIFT: 0xc (12)
TO_PHYS_MASK: 0xffffffff (536870911)
RAM_OFFSET: 0x8000000 (134217728)
MAXPFN: 0x1ffff (131071)
MAXPHYS: 0xffffffff (536870911)
USIZE: 0x1 (1)
EXTUSIZE: 0x1 (1)
UPGIDX: 0xffffffffffffffff
KSTKIDX: 0x0 (0)
EXTSTKIDX: 0x1 (1)
KUBASE: 0x0 (0)
KUSIZE: 0x80000000
KOBASE: 0x80000000
KOSIZE: 0x20000000 (536870912)
KIBASE: 0xa0000000
K1SIZE: 0x20000000 (536870912)
K2BASE: 0xc0000000
K2SIZE: 0x20000000 (536870912)
KERNSTACK: 0xffffd000
KERNELSTACK: 0xffffc000
KEXTSTACK: 0xffffb000
KPTEBASE: 0xff800000
KPTE_USIZE: 0x200000000
KPTE_SHDUBASE: 0xfffffffffff800000
MAPPED_RO_BASE: 0x0 (0)
MAPPED_RW_BASE: 0x0 (0)
MAPPED_PAGE_SIZE: 0x0 (0)

icrash pdeinfo_s data fields

PFN_MASK: 0x3ffffc0 (67108800)
PFN_SHIFT: 0x6 (6)
PDE_PG_VR: 0x2 (2)
PG_VR_SHIFT: 0x1 (1)

Appendix F-6.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

PDE_PG_G: 0x1 (1)
PG_G_SHIFT: 0x0 (0)
PDE_PG_M: 0x4 (4)
PG_M_SHIFT: 0x2 (2)
PDE_PG_N: 0x10 (16)
PG_N_SHIFT: 0x0 (0)
PDE_PG_SV: 0x40000000 (1073741824)
PG_SV_SHIFT: 0x1e (30)
PDE_PG_D: 0x80000000
PG_D_SHIFT: 0x1f (31)
PDE_PG_EOP: 0x40000000 (67108864)
PG_EOP_SHIFT: 0x1a (26)
PDE_PG_NR: 0x18000000 (402653184)
PG_NR_SHIFT: 0x1b (27)
PDE_PG_CC: 0x38 (56)
PG_CC_SHIFT: 0x3 (3)

icrash callback_s data fields

SYM_ADDR: 0x10014870 (268519536)
STRUCT_LEN: 0x10014908 (268519688)
MEMBER_OFFSET: 0x10014980 (268519808)
MEMBER_SIZE: 0x10014a00 (268519936)
MEMBER_BITLEN: 0x10014a88 (268520072)
MEMBER_BASEVAL: 0x10014ab0 (268520112)
BLOCK_ALLOC: 0x10014b40 (268520256)
BLOCK_FREE: 0x10014ba8 (268520360)
PRINT_ERROR: 0x1009e678 (269084280)

icrash global_s data fields

PROGRAM: icrash
FLAGS: 0x0 (0)
DUMPCPU: 0x0 (0)
DUMPKTHREAD: 0x0 (0)
DEFKTHREAD: 0x0 (0)
HWGRAPH: 0xc0002000
ACTIVEFILES: 0x0 (0)
DUMPPROC: 0x0 (0)
ERROR_DUMPBUF: 0x0 (0)
KPTBL: 0x8838c000
LBOLT: 0x88337af4
MLINFOLIST: 0x885dc2a0
NODEPDAINDR: 0x88338010

Appendix F-6.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

PDAINDR: 0x88337488
PIDACTIVE: 0x88338090
PIDTAB: 0xc0026000
PFDAT: 0x8827c128
PUTBUF: 0x8839bc00
STHREADLIST: 0x8835e710
STRST: 0x0 (0)
TIME: 0x88337af0
XTHREADLIST: 0x883575d8
PIDTABSZ: 0x1a8 (424)
PID_BASE: 0x0 (0)
DUMPREGS: 0x0 (0)
KPTBLP: 0x0 (0)
HWGRAPH: 0x10ab1000 (279646208)

O2000 system dump (IRIX 6.5 beta)

```
>> kerninfo
icrash coreinfo_s data fields

CORE TYPE: corefile
PANIC_TYPE: 0x0 (0)
COREFILE: /ptmp/mix/samples/vmcore.15.comp
NAMELIST: /ptmp/mix/samples/unix.15
ICRASHDEF: (null)
CORE_FD: 0x3 (3)
DUMP_HDR: CrshDump
RW_FLAG: 0x0 (0)

icrash sysinfo_s data fields

UTSNAME: IRIX64
IP: 0x1b (27)
PHYSMEM: 0x5000 (20480)
NUMCPUS: 0x6 (6)
MAXCPUS: 0x6 (6)
NLBENTRIES: 0x40 (64)
TLBENTRYSZ: 0x18 (24)
TLBDUMPSIZE: 0x608 (1544)
NUMNODES: 0x3 (3)
MAXNODES: 0x3 (3)
MASTER_NASID: 0x0 (0)
NASID_SHIFT: 0x20 (32)
SLOT_SHIFT: 0x1b (27)
PARCEL_SHIFT: 0x0 (0)
PARCEL_BITMASK: 0x0 (0)
PARCELS_PER_SLOT: 0x0 (0)
SLOTS_PER_NODE: 0x20 (32)
MEM_PER_NODE: 0x100000000
MEM_PER_SLOT: 0x8000000 (134217728)
MEM_PER_BANK: 0x20000000 (536870912)
NASID_BITMASK: 0xff (255)
SLOT_BITMASK: 0x1f (31)
SYSTEMSIZE: 0x0 (0)
SYSTEMSIZE: 0x0 (0)
END: 0x0 (0)
```

icrash kerninfo_s data fields

```
IRIX_REV: 0x5 (5)
SYSSEGSZ: 0x2800 (10240)
NPROCS: 0x528 (1320)
PTRSZ: 0x40 (64)
REGSZ: 0x8 (8)
PAGESZ: 0x4000 (16384)
NBFW: 0x8 (8)
NBFC: 0x4000 (16384)
NCPS: 0x800 (2048)
NBPS: 0x2000000 (33554432)
PNUMSHIFT: 0xe (14)
TO_PHYS_MASK: 0xffffffff
RAM_OFFSET: 0x0 (0)
MAXPFN: 0x3fffffff (67108863)
MAXPHYS: 0xffffffff
USIZE: 0x1 (1)
EXTUSIZE: 0x0 (0)
UPGIDX: 0xfffffffffffffff
KSTKIDX: 0x0 (0)
EXTSTKIDX: 0x0 (0)
KUBASE: 0x0 (0)
KUSIZE: 0x1000000000
KOBASE: 0xa800000000000000
KOSIZE: 0x1000000000
K1BASE: 0x9600000000000000
K1SIZE: 0x1000000000
K2BASE: 0xc000000000000000
K2SIZE: 0xff80000000
KERNSTACK: 0xfffffffffff000
KERNELSTACK: 0xfffffffffff8000
KEXTSTACK: 0x0 (0)
KPTEBASE: 0xc0000fc000000000
KPTE_USIZE: 0x800000000000
KPTE_SHDUBASE: 0xc00007c000000000
MAPPED_RO_BASE: 0xc000000000000000
MAPPED_RW_BASE: 0xc000000001000000
MAPPED_PAGE_SIZE: 0x1000000 (16777216)
```

icrash pdeinfo_s data fields

```
PFN_MASK: 0xffffffff00
PFN_SHIFT: 0x8 (8)
PDE_PG_VR: 0x2 (2)
PG_VR_SHIFT: 0x1 (1)
```

Appendix F-7.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
PDE_PG_G: 0x1 (1)
PG_G_SHIFT: 0x0 (0)
PDE_PG_M: 0x4 (4)
PG_M_SHIFT: 0x2 (2)
PDE_PG_N: 0x28 (40)
PG_N_SHIFT: 0x0 (0)
PDE_PG_SV: 0x1000000000
PG_SV_SHIFT: 0x20 (32)
PDE_PG_D: 0x2000000000
PG_D_SHIFT: 0x21 (33)
PDE_PG_EOP: 0x0 (0)
PG_EOP_SHIFT: 0xfffffffffffffff
PDE_PG_NR: 0x4000000000
PG_NR_SHIFT: 0x22 (34)
PDE_PG_CC: 0x38 (56)
PG_CC_SHIFT: 0x3 (3)
```

icrash callback_s data fields

```
SYM_ADDR: 0x10014880 (268519552)
STRUCT_LEN: 0x10014920 (268519712)
MEMBER_OFFSET: 0x100149a0 (268519840)
MEMBER_SIZE: 0x10014a20 (268519968)
MEMBER_BITLEN: 0x10014ab0 (268520112)
MEMBER_BASEVAL: 0x10014ae0 (268520160)
BLOCK_ALLOC: 0x10014b70 (268520304)
BLOCK_FREE: 0x10014be0 (268520416)
PRINT_ERROR: 0x1009f760 (269088608)
```

icrash global_s data fields

```
PROGRAM: icrash/icrash
FLAGS: 0x0 (0)
DUMP_CPU: 0x4 (4)
DUMP_KTHREAD: 0xa800000103897000
DEFKTHREAD: 0xa800000103897000
HWGRAPH: 0xa80000000055a000
ACTIVEFILES: 0x0 (0)
DUMPPROC: 0xa80000010261b000
ERROR_DUMPBUF: 0x0 (0)
KPTBL: 0xa800000000540000
LBOLT: 0xc00000000144d284
MLINFOLIST: 0xa80000010084efa0
NODEPDAINDR: 0xc0000000014cf9d0
```

Appendix F-7.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
PDAINDR: 0xc00000000144e5a0
PIDACTIVE: 0xc0000000014d0788
PIDTAB: 0xa800000000f14000
PFDAT: 0x0 (0)
PUTBUF: 0xa80000000051b000
STHREADLIST: 0xc0000000014cd370
STRST: 0x0 (0)
TIME: 0xc00000000144d280
XTHREADLIST: 0xc000000001482a70
PIDTABSZ: 0x528 (1320)
PID_BASE: 0x0 (0)
DUMPREGS: 0x10e627d8 (283518936)
KPTBLP: 0x10e79008 (283611144)
HWGRAPH: 0x10e77000 (283602944)
```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix F-7.c

O2000 live system (flurry; IRIX 6.5 beta)

```
>> kerninfo
icrash coreinfo_s data fields

CORE_TYPE: /dev/kmem
PANIC_TYPE: 0x0 (0)
COREFILE: /dev/mem
NAMELIST: /unix
ICRASHDEF: (null)
CORE_FD: 0x4 (4)
DUMP_HDR: (null)
RW_FLAG: 0x2 (2)

icrash sysinfo_s data fields

UTSNAME: IRIX64
IP: 0x1b (27)
PHYSMEM: 0x23c000 (2342912)
NUMCPUS: 0x80 (128)
MAXCPUS: 0x80 (128)
NTLBENTRIES: 0x0 (0)
TLBENTRYSZ: 0x0 (0)
TLBDUMPSIZE: 0x0 (0)
NUMNODES: 0x40 (64)
MAXNODES: 0x40 (64)
MASTER_NASID: 0x0 (0)
NASID_SHIFT: 0x20 (32)
SLOT_SHIFT: 0x1b (27)
PARCEL_SHIFT: 0x0 (0)
PARCEL_BITMASK: 0x0 (0)
PARCELS_PER_SLOT: 0x0 (0)
SLOTS_PER_NODE: 0x20 (32)
MEM_PER_NODE: 0x100000000
MEM_PER_SLOT: 0x8000000 (134217728)
MEM_PER_BANK: 0x20000000 (536870912)
NASID_BITMASK: 0xff (255)
SLOT_BITMASK: 0x1f (31)
SYSTEMSIZE: 0x0 (0)
SYSTEMSIZE: 0x0 (0)
END: 0x0 (0)

icrash kerninfo_s data fields
```

TR-IKI rev 0.7b SGI Proprietary

22jul1998

Appendix F-8

```
IRIX_REV: 0x5 (5)
SYSSEGSZ: 0x11e000 (1171456)
NPROCS: 0x5688 (22152)
PTRSZ : 0x40 (64)
REGSZ : 0x8 (8)
PAGESZ : 0x4000 (16384)
NBPW : 0x8 (8)
NBPC : 0x4000 (16384)
NCPS : 0x800 (2048)
NBPS : 0x2000000 (33554432)
PNUMSHIFT : 0xe (14)
TO_PHYS_MASK : 0xffffffff
RAM_OFFSET : 0x0 (0)
MAXPFN : 0x3fffffff (67108863)
MAXPHYS : 0xffffffff
USIZE : 0x1 (1)
EXTUSIZE : 0x0 (0)
UPGIDX : 0xfffffffffffffff
KSTKIDX : 0x0 (0)
EXTSTKIDX : 0x0 (0)
KUBASE : 0x0 (0)
KUSIZE : 0x10000000000
KOBASE : 0xa800000000000000
KOSIZE : 0x10000000000
K1BASE : 0x9600000000000000
K1SIZE : 0x10000000000
K2BASE : 0xc000000000000000
K2SIZE : 0xfff80000000
KERNSTACK : 0xffffffffffffc000
KERNELSTACK : 0xffffffffffff8000
KEXTSTACK : 0x0 (0)
KPTEBASE : 0xc0000fc000000000
KPTE_USIZE : 0x80000000000
KPTE_SHDUBASE : 0xc00007c000000000
MAPPED_RO_BASE : 0xc000000000000000
MAPPED_RW_BASE : 0xc000000001000000
MAPPED_PAGE_SIZE : 0x1000000 (16777216)
```

icrash pdeinfo_s data fields

```
PFN_MASK: 0xffffffff00
PFN_SHIFT: 0x8 (8)
PDE_PG_VR: 0x2 (2)
PG_VR_SHIFT: 0x1 (1)
```

Appendix F-8.a

22jul1998

TR-IKI rev 0.7b SGI Proprietary

```
PDE_PG_G: 0x1 (1)
PG_G_SHIFT: 0x0 (0)
PDE_PG_M: 0x4 (4)
PG_M_SHIFT: 0x2 (2)
PDE_PG_N: 0x28 (40)
PG_N_SHIFT: 0x0 (0)
PDE_PG_SV: 0x1000000000
PG_SV_SHIFT: 0x20 (32)
PDE_PG_D: 0x2000000000
PG_D_SHIFT: 0x21 (33)
PDE_PG_EOP: 0x0 (0)
PG_EOP_SHIFT: 0xfffffffffffffff
PDE_PG_NR: 0x4000000000
PG_NR_SHIFT: 0x22 (34)
PDE_PG_CC: 0x38 (56)
PG_CC_SHIFT: 0x3 (3)
```

icrash callback_s data fields

```
SYM_ADDR: 0x10014880 (268519552)
STRUCT_LEN: 0x10014920 (268519712)
MEMBER_OFFSET: 0x100149a0 (268519840)
MEMBER_SIZE: 0x10014a20 (268519968)
MEMBER_BITLEN: 0x10014ab0 (268520112)
MEMBER_BASEVAL: 0x10014ae0 (268520160)
BLOCK_ALLOC: 0x10014b70 (268520304)
BLOCK_FREE: 0x10014be0 (268520416)
PRINT_ERROR: 0x1009f760 (269088608)
```

icrash global_s data fields

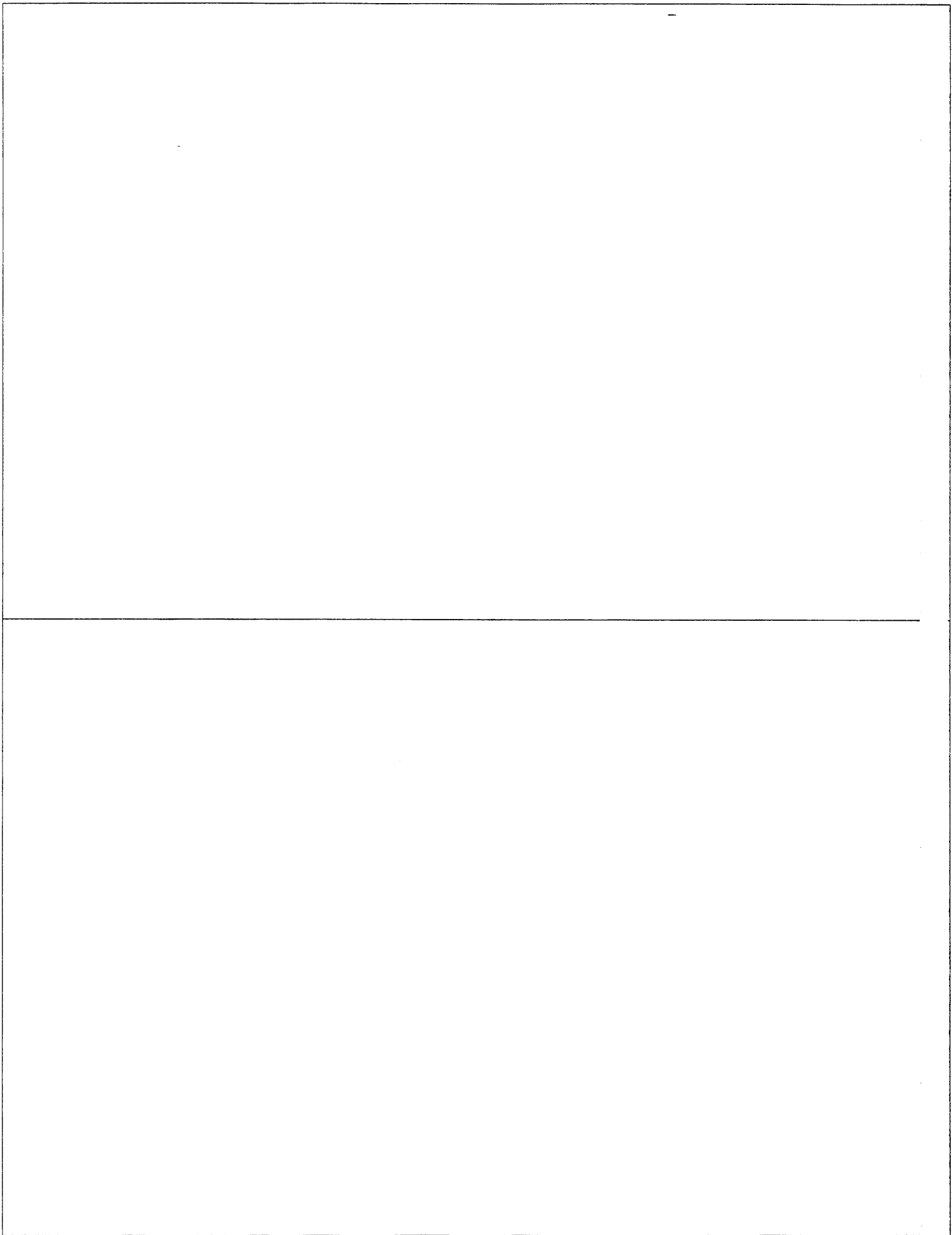
```
PROGRAM: icrash
FLAGS: 0x0 (0)
DUMPCPU: 0x0 (0)
DUMPKTHREAD: 0x0 (0)
DEFKTHREAD: 0x0 (0)
HWGRAPHP: 0xa800000001370000
ACTIVEFILES: 0x0 (0)
DUMPPROC: 0x0 (0)
ERROR_DUMPBUF: 0x0 (0)
KPTBL: 0xa80000000071c000
LBOLT: 0xc00000000145d80c
MLINFOLIST: 0xa8000008009240c0
NODEPDAINDR: 0xc0000000014bd288
```

Appendix F-8.b

22jul1998

TR-IKI rev 0.7b SGI Proprietary

PDAINDR: 0xc0000000145eaa0
PIDACTIVE: 0xc000000014be040
PIDTAB: 0xc00000003ab4000
PFDAT: 0x0 (0)
PUTBUF: 0xa800000005c2c00
STHREADLIST: 0xc000000014bab08
STRST: 0x0 (0)
TIME: 0xc0000000145d808
XTHREADLIST: 0xc000000014b6060
PIDTABSZ: 0x5688 (22152)
FID_BASE: 0x0 (0)
DUMPREGS: 0x0 (0)
KPTBLF: 0x0 (0)
HWGRAPH: 0x10dff000 (283111424)



Appendix G: How to get a core dump from your Indy (2 methods)

How to get a core dump from your Indy (2 methods)

There are two ways to force a core dump file on your Indy.

Both of these require you to have root privileges on the machine.

METHOD 1 : Change a systune parameter

- only available as of 6.5
- means you always get a core file when you push the power button from then on, until you change the systune parameter again

METHOD 2: Use the debugger to damage /dev/kmem.

By default, the core dump file and related files will be placed in the `/usr/adm/crash` directory (which is the same as the `/var/adm/crash` directory).

METHOD 1 - change systune parameter

The SYSTUNE parameter "`power_button_changed_to_crash_dump`" controls whether pushing the restart button causes a core dump to be taken or not.

You must have root privileges on the machine to change SYSTUNE settings.

1. "su" to "root".
2. Use "grep" to search through your current SYSTUNE settings for the current value of the parameter "`power_button_changed_to_crash_dump`".

A method is shown below:

```
myindy# systune | grep -i power
power_button_changed_to_crash_dump = 0 (0x0)
powerfail_routerwar = 1 (0x1)
powerfail_cleanio = 0 (0x0)
```

3. Set the parameter value to anything but 0.

An example is shown below:

```
myindy# systune power_button_changed_to_crash_dump 1
```

4. The system will respond with the following two lines. Type "y" as your answer to the question, as shown below:

```
power_button_changed_to_crash_dump = 0 (0x0)
```

```
Do you really want to change power_button_changed_to_crash_dump to 1 (
```

METHOD 2 - damage /dev/kmem

You must have root privileges on the machine to change use the **dbx** debugger on kernel memory.

1. "su" to "root".
2. Type the following:

```
dbx -k /unix /dev/kmem
```

3. at the "dbx>" prompt, type:

```
p &fork
```

4. That gave you the starting address of the fork routine, 0x____something_____.

An example, and the output generated, are shown below:

```
(dbx) p &fork  
0x88196c44
```

5. Now overwrite that memory location, damage the kernel's copy of the "fork" code, and confuse the system, by changing the starting address of the "fork" routine to "0".

An example is shown below, using the address displayed in the previous step:

```
assign ((int *) 0x88196c44 ) = 0
```

NOTE ! As ***SOON*** as you hit "return" after typing the above, your Indy will panic. So have

everything else already set up first, because you will not get a chance to exit from the debugger, or do anything else.

