

Kernel Internals

AIX Worldwide Technology Transfer

Notes: Kernel Internals

First a few statements concerning the format of this workshop. AIX Worldwide Technology Transfer provides AIX T3 classes. T3 stands for Teach The Teacher and means that these classes are designed for people who have a need for the technical material delivered, as well as the responsibility to deliver education in the respective area. T3 classes and class material are designed so that the class attendees with the right set of prerequisite skills will be able to pick up the material and reteach the workshop to their own audience. This goal is partly accomplished through the form of this class material: The material is made up out of a set of foils. Each foil will be accompanied by a page of notes. The foils highlight the most important facts of a certain technical issue. The notes deliver additional detailed information for the class attendees as well as notes targeted towards the instructor and give hints and tips on how to present a certain subject matter and related information that might be valuable to point out to students.

Contents

- Chapter 1: Kernel Overview
- Chapter 2: Memory Management
- Chapter 3: Process Management
- Chapter 4: Kernel Extensions
- Chapter 5: File Systems
- Chapter 6: Loader
- Chapter 7: AIX boot
- Chapter 8: Logical Volume Manager
- Chapter 9: System Dump
- Programming Examples
- Exercises

Notes: Content

The following skills/classes are pre-requisites to this class: knowledge of C programming and basic operating systems concepts.

Chapter one provides a general introduction to the kernel, RS/6000 hardware, and exception handling.

Chapter two provides an in-depth discussion of memory management.

Chapter three provides an in-depth discussion of process and thread management.

Chapter four provides an in-depth discussion of kernel extension issues.

Chapter five provides an in-depth discussion of file systems.

Chapter six provides an in-depth discussion of the loader.

Chapter seven provides an in-depth discussion of system boot.

Chapter eight provides an in-depth discussion of the Logical Volume Manager (LVM).

Chapter nine provides an in-depth discussion of system dump.

A separate **Programming Examples Book** is available containing programming examples.

A separate **Lab And Exercise Book** is available containing quizzes and hands-on lab-sessions.

Special Notices

This publication/presentation was produced in the United States. IBM may not offer the products, programs, services or features discussed herein in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the products, programs, services and features available in your area. Any reference to an IBM product, program, service or feature is not intended to state or imply that only IBM's product, program, service or feature may be used. Any functionally equivalent product, program, service or feature that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, service or feature.

Information in this presentation concerning non-IBM products was obtained from the suppliers of these products, published announcement material or other publicly available sources. Sources for non-IBM list prices and performance numbers are taken from publicly available information including D.H. Brown, vendor announcements, vendor WWW Home Pages, SPEC Home Page, GPC (Graphics Processing Council) Home Page and TPC (Transaction Processing Performance Council) Home Page. IBM has not tested these products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this presentation. The furnishing of this presentation does not give you any license to these patents. Send license inquires, in writing, to IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your IBM local Branch Office or IBM Authorized Reseller for the full text of a specific Statement of General Direction.

The information contained in this presentation has not been submitted to any formal IBM test and is distributed AS IS. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. The use of this information or the implementation of any techniques described herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Special Notices (Continued)

The information contained in this document represents the current views of IBM on the issues discussed as of the date of publication. IBM cannot guarantee the accuracy of any information presented after the date of publication.

All prices shown are IBM's suggested list prices; dealer prices may vary.

IBM products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Any performance data contained in this document was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements quoted in this presentation may have been made on development-level systems. There is no guarantee that these measurements will be the same on generally-available systems. Some measurements quoted in this presentation may have been estimated through extrapolation. Actual results may vary. Users of this presentation should verify the applicable data for their specific environment.

The following terms are trademarks of International Business Machines Corporation in the United States and/or other countries: AIX, PowerPC, RS/60000, SP.

Microsoft, Windows, Windows NT and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/ Open Company, Limited. Other company, product and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

© Copyright INTERNATIONAL BUSINESS MACHINES CORPORATION 1998

Kernel Overview

Notes: Kernel Overview

This page intentionally left blank.

Outline

- Kernel Description
- Hardware Overview
  - *Registers*
  - *Instruction Set*
- 64-bit Hardware
- Multiprocessor Hardware
- Exceptions and Interrupts
  - *Exceptions vs Interrupts*
  - *Interrupt Handling*
  - *State Save*

Notes: Outline

This page intentionally left blank.

Kernel Description

Notes: Kernel Description

This page intentionally left blank.

Kernel Description

- **The Kernel is a linked object file similar to a user application**
- **Functions of the kernel:**
  - ***Provide mechanism for creation and deletion of processes.***
  - ***Provide CPU scheduling, memory management, and device management.***
  - ***Provide synchronization tools.***
  - ***Provide communication tools for process communication.***

Notes: Kernel Description

In AIX as in other UNIX-like systems, the kernel is a linked object file similar to a user application.

Functions of the Kernel are:

To provide a mechanism for the creation and deletion of processes.

To provide CPU scheduling, memory management, and device management.

To provide synchronization tools.

To provide communication tools for process communication.

Distinguishing characteristics of the AIX kernel include:

A Preemptable Kernel

Better hooks for Kernel Extensions

A Pageable Kernel

Preemptable Kernel:

In AIX, the kernel can be in the middle of a system call and be preempted. The preemption could cause a context switch that causes an entirely new thread of execution inside the kernel.

Better hooks for Kernel Extensions:

Kernel extensions are dynamically loadable in AIX

Pageable Kernel:

In AIX, part of the kernel is pageable. In other UNIX, the entire kernel must be loaded and pinned in memory.

System Kernel Diagram

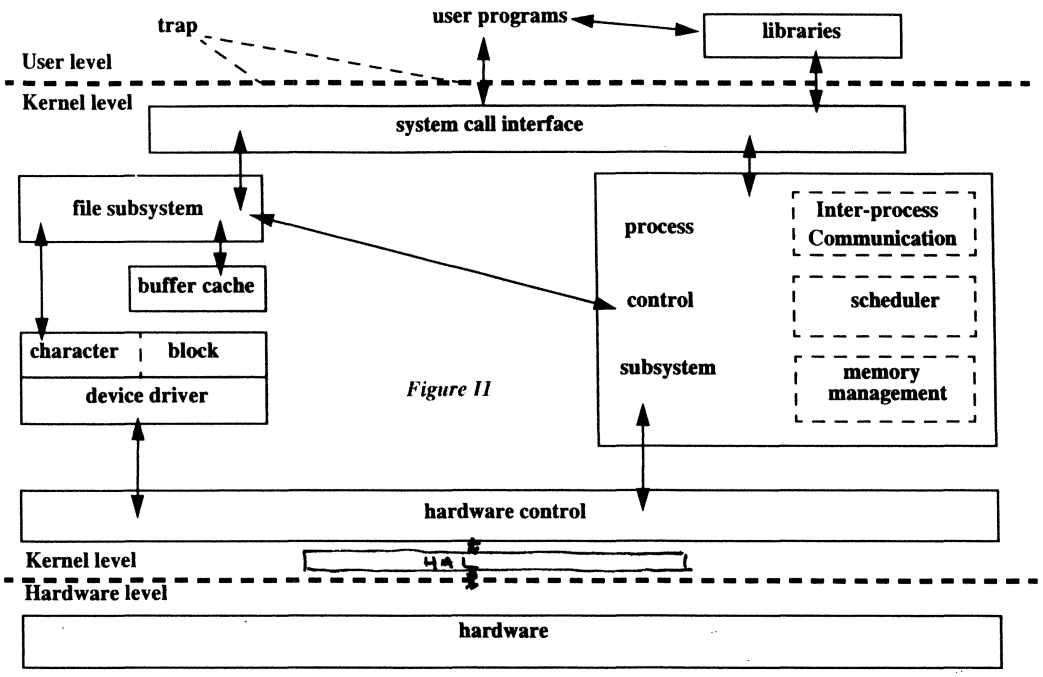


Figure 11

Notes: System Kernel Diagram

This page intentionally left blank.

Kernel Services

- Process management manages CPU usage.
- Memory management allows processes to share memory.
- Device drivers provide access to devices.
- The Filesystem provides abstraction of file objects.
- Kernel Extension enables user to add additional services.

Notes: Kernel Services

In AIX, the kernel provides the ability to share system resources simultaneously among many processes and users. Many other non-UNIX operating systems allow only single access to the machine resources.

The kernel shares resources by rules contained in the kernel code. The most important resources on the computer are CPU, memory and devices.

Kernel Protection Domain

- **Kernel Protection Domain (Kernel-Mode)**
  - ***Code running in this domain has read/write access to global kernel space and access to kernel data in the process private region when running within the process context.***
  - ***Code in interrupt handlers, the base kernel, and kernel extensions run in kernel mode.***
  - ***Programs running in kernel-mode access user data via kernel services.***
  - ***Programs running in kernel-mode:***
    - ***can use kernel services***
    - ***can access global system data***
    - ***are exempt from all security restraints***
    - ***run in the processor privileged state***

Notes: Kernel Protection Domain

Programs that run in the kernel protection domain include interrupt handlers, the base kernel, and kernel extensions (including device drivers). This protection domain implies that code runs in the kernel execution mode and has the following access:

Read/write access to global kernel address space

Read/Write access to the kernel data in the process private region when running in the process context.

User data within the process address space must be accessed using kernel services. Programs running in this protection domain can affect the execution environment of all programs.

All kernel extensions run in the kernel protection domain as described above. The use of a system call by a user-mode process allows a kernel function to be called from user-mode. Access to functions that directly or indirectly invoke system calls is typically provided by programming libraries providing access to operating system functions.

User Protection Domain

- **User Protection Domain (User-Mode)**
  - *Code runs in user execution mode*
  - *User-mode access:*
    - *Read/write access to the user data in the process private region*
    - *Read access to the user text and shared library text region*
    - *Read/write access to the shared library data region*
    - *Access to shared memory regions using the shared memory functions*
  - *User-Mode privileges:*
    - *Can only affect its own execution environment and runs in the processor unprivileged state*

Notes: User Protection Domain

Programs running in the user protection domain do not have access to the kernel or kernel data segments except indirectly through the use of system calls. A program in this protection domain can only affect its own execution environment and runs in the processor unprivileged state.

Hardware Overview

Notes: Hardware Overview

This page intentionally left blank.

Hardware Overview

- **POWER and 32-bit PowerPC processors all have the following features in common:**
  - *User registers*
    - *32 general-purpose integer registers, each 32 bits wide (GPRs)*
    - *32 floating-point registers, each 64 bits wide (FPRs)*
    - *A 32-bit Condition Register (CR)*
    - *A 32-bit Link Register (LR)*
    - *A 32-bit Count Register (CTR)*
    - *etc.*

Notes: Hardware Overview

This page intentionally left blank.

Hardware Overview

- *System Registers*
  - *16 Segment Registers (SRs)*
  - *A Machine State Register (MSR)*
  - *A Data Address Register (DAR)*
  - *Two Save and Restore Registers (SRRs)*
  - *4 special purpose (SPRG) registers (PowerPC only)*
  - *etc.*
- *All instructions are 32 bits long*
- *Support for cache memory*
- *Etc.*

Notes: Hardware Overview

This page intentionally left blank.

User Register Usage

- **General Purpose Registers (GPRs) (often just called Rs) used for loads, stores, and integer calculations**
  - ***No memory-to-memory operations provided - always go through registers***
- **Floating Point Registers (FPRs) used in floating-point operations**
- **Condition Register (CR)**
  - ***Contains bits set by the results of compare instructions***
  - ***Treated as 8 4-bit registers***
  - ***Bits are for less-than, greater-than, equal, and overflow***

Notes: User Register Usage

This page intentionally left blank.

User Register Usage

- **Link Register (LR) is set by some branch instructions**
  - ***Points to instruction immediately after branch***
  - ***Used in subroutine calls to find out where to return to***
- **Count Register (CTR) has two uses**
  - ***Can be decremented, tested, and used to decide whether to take a branch, all from one branch instruction***
  - ***Can contain the target address for a branch instruction***

Notes: User Register Usage

This page intentionally left blank.

System Register Usage

- **Segment Registers (SRs) are used in virtual addressing mode in conjunction with virtual addresses to provide a large virtual address space**
- **The MSR controls many of the current operating characteristics of the processor**
  - ***Privilege Level (Supervisor vs. Problem or Kernel vs. User)***
  - ***Addressing modes (virtual vs. real)***
  - ***Interrupt enabling***
  - ***Little-endian vs. Big-endian mode***
  - ***etc.***

Notes: System Register Usage

This page intentionally left blank.

AIX in BigEndian

System Register Usage

- The DAR contains the memory address that caused the last memory-related exception
- SRRs used to save information when an interrupt occurs
  - *SRR0 points to the instruction that was running when the interrupt occurred*
  - *SRR1 contains the contents of the MSR when the interrupt occurred*
- SPRGs are used for general operating system purposes, requiring per-processor temporary storage
  - *Fast state saves*
  - *Support multi-processing environments*

Notes: System Register Usage

This page intentionally left blank.

Instruction Set

- **A single instruction generally modifies only one register or one memory location**
  - ***Exceptions: "multiple" and "update" operations***
- **The format is:**
  - ***An opcode mnemonic***
  - ***An optional set of option bits***
    - ***These two may be combined into an "extended mnemonic"***
  - ***0, 1, 2, or 3 registers***
  - ***0 or 1 memory locations, expressed as an offset added to/subtracted from a register***
    - ***e.g.*** `24(r3)` ***means the address in r3 + 24***
- **General Purpose Registers named "r0" - "r31"**

Notes: Instruction Set

In case you are not familiar with the term, *mnemonic* is defined in the dictionary as, "a device or code intended to assist memory." The term here applies to the 1 to 6 character codes that are used to indicate the opcode for an instruction to an assembler.

POWER vs. PowerPC Mnemonics

- **Although most instructions are the same, the mnemonics for POWER and PowerPC are often different**
  - ***POWER mnemonics are generally simpler and shorter***
    - `l` - ***"Load" - loads a 32-bit word into a register***
  - ***PowerPC mnemonics are longer, but more explicit***
    - `lwz` - ***"Load Word and Zero" - same thing***
- **Differences are because PowerPC was developed with 64-bit in mind**
  - ***In 64-bit, a "`W(ord)`" is 32 bits only, regardless of the machine mode***
  - ***64-bit values are called "`D(ouble)`"***
- **Note: the actual opcodes generated by the assembler for these instructions are identical**

Notes: POWER vs PowerPC Mnemonics

This page intentionally left blank.

Register-Register Operations

- **Always have at least 2 registers listed**
  - *First is target for result of instruction*
  - *Others provide input to operation*
- **"Immediate" operations shown as register with an offset**
  - *"Immediate" means that a constant value is involved*
    - *Value built right into the instruction*
- **Examples**
  - `or  r3, r4, r5  # Logical ORs r4 and r5, result into r3`
  - `addi  r1, 0x48(r1)  # Adds 0x48 to r1, result into r1`

Notes: Register-Register Operations

This page intentionally left blank.

Register-Memory Operations

- **Always one register and one memory location**
  - ***Register always listed first***
- **Size of memory location is specified in the opcode**
  - `b` = ***byte (8 bits)***
  - `h` = ***halfword (16 bits)***
  - `w` = ***word (32 bits)***
  - `d` = ***doubleword (64 bits)***
- **All opcodes beginning with "`l`" are loads**
- **All opcodes beginning with "`st`" are stores**

Notes: Register-Memory Operations

This page intentionally left blank.

Register-Memory Operations

- **Examples:**
  - `lwz r31, 4(r30)` - *Loads 32 bits from address 4+r30 into r31*
    - *High 32-bits cleared on 64-bit processor*
  - `std r3, -8(r29)` - *Stores 64 bits from r3 to address r29 - 8*
    - *Invalid operation on 32-bit processor*
  - `lbz r0, 0x27(r1)` - *Loads 8 bits from address 27+r1 into r0*
    - *Top 24/56 bits are cleared*
  - `sth r3, 0x56(r1)` - *Stores low 16 bits from r3 to address 0x56+r1*

Notes: Register-Memory Operations

Only PowerPC mnemonics are shown on this page, to help illuminate the differences among the actual instruction mnemonics. There are equivalent POWER mnemonics for all these instructions except the 64-bit ones. Notice that the load instructions also have a "z" in their mnemonics. The "z" stands for "zero," and is intended to make clear that these instructions clear any bits in the target register that were not actually copied from memory.

In case you were wondering, there are load instructions without the "z". lwa and lha are "algebraic" loads. This means that the value being loaded is sign-extended to fill out the rest of the register. This is used when loading a signed value - if a halfword had a negative value, lhz would make it a positive, but lha would preserve the value's "negativeness."

Compare Instructions

- 4 variations, all beginning with "cmp"
- Compare two values
  - *Register and register, or*
  - *Register and immediate value (i.e. constant value)*
- Result of comparison is placed in the specified CR
- Bits set in CR:
  - *LT = less than*
  - *GT = greater than*
  - *EQ = equal*
  - *OV = overflow (a.k.a. carry bit)*

Notes: Compare Instructions

This page intentionally left blank.

Branch Instructions

- All instructions beginning with "b" are branches
- Change the address for the next instruction to be run
- Three addressing modes:
  - *Absolute - goes to an explicit address*
  - *Relative - target address is an offset from current instruction address*
  - *Register - Only two registers can contain a branch target*
    - *Count (CTR) and Link (LR)*
- Can be conditional
  - *Depends upon whether option bit matches specified bit in CR*
  - *Can specify which CR to use*
    - *CR0 assumed unless otherwise specified*

Notes: Branch Instructions

The conditional branch instruction is central to any computer architecture. It is what separates calculators from computers. It provides the ability to make a decision based upon some data and alter the sequence of program execution as a result of that decision. Without it, there could be no IF tests in programs.

However, most architectures (including POWER and PowerPC) avoid putting comparisons directly into their branch instructions (to keep things simple). They provide compare instructions that set "condition bits." These bits are what are used on branch instructions to make the actual decision.

Branch Instructions

- **Extended mnemonics defined by assembler to cover most combinations**
- **Can only use one CR bit per branch**
  - ***But can negate (i.e. logical NOT) the test***
- **Examples**
  - `blt 0x38c00` - ***Branches to address 38c00 if LT bit is on in CR0***
  - `bge cr3, 0x48950` - ***Branches if LT bit is off in CR3***
  - `bnelr cr7` - ***Branches to address in LR if EQ bit is off in CR7***
  - `blea cr2, 0x3600` - ***Branches to absolute address 0x3600 if GT bit is off in CR2***

Notes: Branch Instructions

The assembler (and crash's disassembler) provides extended mnemonics that combine a type of branch and the condition register bit that determines whether the branch is taken. Another bit in the branch opcode determines whether the CR bit must be on or off for the branch to take place. This bit is also incorporated into the extended mnemonics (the "not" versions of the branches). For maximum flexibility, the assembler usually also allows you to specify the "not" cases as the logically-opposite case. For example, `bnl` (branch not less than) can also be written as `bge` (branch greater than or equal to). Either case is still saying, "branch if the LT bit is turned off."

Trap Instructions

- **Most mnemonics beginning with "t" are traps**
- **Generate a program exception if the specified condition is met**
  - ***Otherwise does nothing***
- **Two forms:**
  - `t` ***or*** `tw` ***- compares two registers, traps if specified comparison is true***
  - `ti` ***or*** `twi` ***- compares register to immediate value instead***
    - ***"`w`" mnemonics are PowerPC indication that these are working on 32-bit values***

Notes: Trap Instructions

This page intentionally left blank.

Trap Instructions

- **Extended mnemonics defined similar to branches**
  - `lt, gt, eq, etc.` ***have same meaning as on branch mnemonics***
- **Examples**
  - `tweq r3, r4` - ***Traps if r3 equals r4***
  - `twnei r31, 0` - ***Traps if r31 is not equal to 0***
- **The two registers may be the same**
  - ***e.g.*** `tweq r3, r3`
  - ***If condition is "eq", then this is an unconditional trap***

Notes: Trap Instructions

Trap instructions are the only instructions in this architecture that perform a comparison and take some action, all in one instruction. They do not set or use condition register bits.

Special Register Operations

- **The Special Purpose Registers (SPRs) can only be copied to or from GPRs**
  - ***mfspr r3, 8 - Copies SPR 8 into r3***
  - ***mtspr 9, r3 - Copies r3 into SPR 9***
- **Extended mnemonics defined to cover common SPRs**
  - ***mflr r3 - Copies the LR (SPR 8) into r3***
  - ***mtctr r3 - Copies r3 into the CTR (SPR 9)***

Notes: Special Register Operations

This page intentionally left blank.

64-bit Hardware

Notes: 64-bit Hardware

This page intentionally left blank.

64-bit Hardware

- **GPRs are 64 bits wide**
  - *Can be used for 64 bit data values*
  - *Can be used with 64 bit load and store instructions*
  - *Can be used for 64 bit arithmetic instructions*
  - *Can store 64 bit addresses*
- **Addresses are 64 bits wide**
  - *Effective addresses can represent roughly 16,000,000,000,000,000,000 bytes*
  - *Physical address allow same amount of physical memory on machine*

Notes: 64-bit Hardware

This page intentionally left blank.

64-bit Hardware

- **Machine Status Register (MSR) bit controls 32-bit or 64-bit execution mode**
  - ***Allows support for 32-bit processes on 64-bit hardware***
  - ***Used by the kernel to run in 32-bit mode in kernel***
  - ***portions of the VMM run in 64-bit mode on 64-bit hardware (to address large tables to represent large virtual memory)***
  - ***32-bit mode on 64-bit hardware looks exactly like 32-bit hardware (ensures binary compatability for 32-bit applications)***
  - ***32-bit instructions use only bottom 32-bits of registers for data or addresses***

Notes: 64-bit Hardware

The only difference between 32-bit hardware and 32-bit mode on 64-bit hardware is that 64-bit instructions are legal in 32-bit mode on 64-bit hardware. The kernel does not save the upper half of the 64-bit registers for 32-bit processes however, so any interrupt that occurs will result in the data in the upper half of registers being lost. Since user mode code can't prevent interrupts from occurring between any two instructions (user code can't disable interrupts), the upper half of registers can't be reliable used in 64-bit user mode code.

Thus 64-bit instructions are allowed by the hardware in 32-bit mode, but are not supported by AIX. (The reason for this is that we would have to waste system resources for saving the state of 32-bit processes if we decided to save the full 64-bit value of registers. There is no significant performance issue because saving 32-bit registers takes the same number of instructions as saving 64-bit registers, though more cache lines may be hit.)

64-bit Hardware

- **64-bit virtual address space is represented with a segment table**
  - ***acts as an in-memory set-associative cache of the most recently used 256 segment number to segment ID mappings***
  - ***current segment table is pointed to with the 64 bit Address Space Register (ASR) register***
- **Segment Lookaside Buffer (SLB) used to cache recently used segment number to segment ID mappings**
  - ***similar to Translation Lookaside Buffer (TLB) for page to frame translations***
  - ***similar to segment table but smaller and faster (on chip, not in memory)***

Notes: 64-bit Hardware

The ASR has a valid bit to indicate that no segment table is valid. This is used in 32-bit mode on 64-bit processors to indicate that the segment table is not being used. The "bridge extensions" are used to load and store "segment registers" instead.

64-bit Hardware

- **IBM "bridge extensions" to PowerPC 64-bit architecture allow segment register operations to work for 32-bit mode**
  - ***allowed kernel to continue to manipulate segment registers***

Notes: 64-bit Hardware

This page intentionally left blank.

Multiprocessor Hardware

Notes: Multiprocessor Machines

This page intentionally left blank.

Symmetric Multiprocessing (SMP)

- **Overcomes uniprocessor (UP) bottlenecks.**
- **Involves using more than one CPU.**
- **Types of Multiprocessors:**
  - *Loosely-coupled MP*
    - ***Has different systems on a communication link with the systems functioning independently and communicating when necessary.***
    - ***The separate systems can access each other's files and maybe even download tasks to the lightly loaded CPU to achieve some load balancing.***
  - *Tightly-coupled MP*
    - ***Uses a single storage shared by the various processors and a single operating system that controls all the processors and system hardware.***

Notes: Symmetric Multiprocessing (SMP)

Multiprocessing:

On **uniprocessor systems**, bottlenecks exist in the form of the address and data bus restricting transfers to one at a time and the other program counter forcing instructions to be executed in strict sequence. Some performance improvement was achieved by constantly improving the speeds of these uniprocessor machines.

Multiprocessing involves more than one CPU. There are several categories of MP systems depending on whether the CPU share resources, have their own resources (like memory, operating system, I/O channels, control units, files and devices), how they are connected (whether in a single machine sharing a single bus or in different machines), whether all processors are functionally equal or some are specialized.

Symmetric Multiprocessing (SMP)

- *Symmetric Multi-Processing*
  - ***All of the processors are functionally equivalent and can perform I/O and computation.***

Notes: Symmetric Multiprocessing (SMP)

This page intentionally left blank.

Multiprocessor Organization

- **Master/slave multiprocessing organization.**
  - ***One processor is designated as the master and the others are the slaves.***
  - ***The master is a general purpose processor and performs input/output as well as computation.***
  - ***The slave processors perform only computation.***
  - ***The processors are considered asymmetric (not equivalent) since only the master can do I/O as well as computation.***
  - ***Utilization of a slave may be poor if the master does not service slave requests efficiently enough.***
  - ***I/O-bound jobs may not run efficiently since only the master does I/O.***

Notes: Multiprocessor Organization

This page intentionally left blank.

Multiprocessor Organization

- **Separate executives organization.**
  - ***Each processor has its own operating system and responds to interrupts from users running on that processor.***
  - ***A process is assigned to run on a particular processor runs to completion.***
  - ***It is possible for some of the processors to remain idle while other processors execute lengthy processes.***
  - ***Some tables are global to the entire system and access to these tables must be carefully controlled.***
  - ***Each processor controls its own dedicated resources, such as files and I/O devices.***

Notes: Multiprocessor Organization

This page intentionally left blank.

Multiprocessor Organization

- **Symmetric multi-processing organization.**
  - ***All of the processors are functionally equivalent and can perform I/O and computation.***
  - ***The operating system manages a pool of identical processors, any one of which may be used to control any I/O devices or reference any storage unit.***
  - ***Conflicts between processors attempting to access the same storage at the same time are ordinarily resolved by hardware.***
  - ***Multiple tables in the kernel can be accessed by different processes simultaneously.***
  - ***Conflicts in access to systemwide tables are ordinarily resolved by software.***
  - ***A process may be run at different times by any of the processors and at any given time, several processors may execute operating system function in kernel-mode.***

Notes: Multiprocessor Organization

This page intentionally left blank.

Multiprocessor Definitions

- **Physical CPU number**
  - ***Physical numbers identify all processors on the system, regardless of their state. Always starts with 0 (zero).***
- **Logical CPU number**
  - ***Logical numbers identify enabled processors only. Always start with 0 (zero).***
- **Master Processor - the default processor**
  - ***The master processor, defined by the value of MP_MASTER in the <sys/processor.h> file, is logical CPU 0.***
  - ***It is not a master processor in the sense of master/slave processing - the term is used only to designate which processor will be the default processor.***
  - ***The master processor is used for funneling.***

Notes: Multiprocessor Definitions

Multi-Processor Definitions:

Physical and Logical CPU number - Processors are identified using either physical numbers or logical numbers. Processor numbering always start with 0 (zero). Physical numbers identify all processors on a system, regardless of their state, whereas logical numbers only identify enabled processors. The Object Data Manager (ODM) names for processors are based on physical numbers with the prefix /proc. The table below illustrates these naming scheme for a three-processor system.

Figure 16

ODM Name	Physical Number	Logical Number	Processor State
/proc0	0	0	Enabled
/proc1	1		Disabled
/proc2	2	1	Enabled

Multiprocessor Definitions

- *In order to run some Uni-Processor device drivers unchanged, their execution had to be "funneled" through one specific processor, which is called the MP master.*

- **Funneling**
  - *Funneled code runs only on the master processor; therefore, the current uniprocessor serialization is sufficient.*
  - *Interrupts for a funneled device driver will be routed to the MP master CPU.*
  - *Funneling is intended to support third party device driver and low-throughput device drivers.*
  - *The base kernel will provide binary compatibility for these device drivers.*
  - *Funneling only works if all references to the device driver are through the device switch table.*

Notes: Multiprocessor Definitions

This page intentionally left blank.

Multiprocessor Definitions

- **MP safe**
  - *The device driver runs on any processor.*
  - *The code for the device driver has been modified to add a code lock to serialize device drivers execution.*
  - *This type of device driver is intended for medium-throughput device drivers.*
- **MP efficient**
  - *The device driver runs on any processor.*
  - *The code for the device driver has been modified to add data locks to serialize the device drivers' accesses to devices and data.*
  - *This type of device driver is intended for high-throughput device drivers.*

Notes: Multiprocessor Definitions

This page intentionally left blank.

Exceptions and Interrupts

Notes: Exceptions and Interrupts

The distinction between the terms "exception" and "interrupt" is often blurred. The bulk of AIX documentation refers to both classes generically as "interrupts," while the hardware documentation (like the PowerPC 60x User's Manuals) makes the distinction. We will try to keep the terms separate.

Interrupts

- Interrupts are asynchronous events that may be generated by the kernel or a device, and "interrupts" the execution of the current process.
- Interrupts usually occur when a process is running and some asynchronous event occurs such as disk I/O completion or a clock tick.
- The event usually has nothing to do with the current running process. The kernel immediately preempts the current running process to handle the interrupt.
- The state of the machine is saved on the stack and the interrupt is handled. The user process has no knowledge that the interrupt occurred.

Notes: Interrupts

Interrupts are one of the major reasons that AIX cannot be a hard real-time system. No guarentee can be made as to how long it may take for some action to occur as it may get interrupted any number of times during the action.

Interrupts

- Interrupts are caused outside the context of a process.
- In general, a process may NOT *decide how to react to the interrupt. Interrupt handlers are kernel code and run without regard to the process unless the nature of the interrupt is to update some process related structure, *statistics, etc..
- A process can disable some or all types of interrupts for short periods.
- The process will safely return to continue execution.
- Some interrupt types can be found in <sys/m_intr.h>

Notes: Interrupts

Most interrupts are not concerned with which process is getting interrupted. The major counterexample is the clock interrupt. This is used to update the run-time statistics for the currently running process.

A process would disable interrupts if the process is manipulating data that is also examined or modified by a device driver interrupt routine. Disabling interrupts ensures that the data will only be used by the interrupt handler when it is in a consistent state.

Exceptions

- Exceptions are synchronous events that are normally caused by the process doing something illegal.
- An exception is a condition caused by a process attempting to perform an action that is not allowed, such as writing to a memory location not owned by the process, or that requires kernel intervention, such as an alignment exception.
- For illegal operations, the kernel traps the offending action and delivers a signal to the process causing the exception or crashes if the process was in kernel mode.
- Exceptions can also be caused by a page fault. A page fault is a reference to a virtual memory location for which the associated real data is not in physical memory.

Notes: Exceptions

The decision to signal the process or crash the machine is based upon what kind of exception occurred and whether the process was executing in user mode or kernel mode.

Exceptions

- Exceptions are caused within the context of a process.
- A process may NOT decide how to react to the exception.
- Exception handlers are kernel code and run without regard to the process except to cleanly handler the exception generated by the process.
- Some exceptions result in the death of the process.
- Some exception types can be found in <sys/m_except.h>

Notes: Exceptions

A process can decide how to respond to the signal generated due to the exception, in certain cases. For example, a process can decide to catch the signal for SIGILL which is delivered when a process in user mode executes an illegal instruction.

Execution Modes

- **User mode vs. Kernel mode**
  - ***User mode - a.k.a. "Problem State"***
    - ***User programs and applications run in this mode***
    - ***Kernel data and global structures protected from access/modification***
  - ***Kernel mode - a.k.a. "Supervisor State"***
    - ***Kernel and kernel extensions run in this mode***
    - ***Can access/modify anything***
    - ***Certain instructions limited to supervisor state only***

Notes: Execution Modes

This page intentionally left blank.

Exceptions

- **Mechanism to change to supervisor state as a result of**
  - ***External signals***
  - ***Program errors***
  - ***Unusual conditions***
  - ***Program requests***
- **Processor receives notice of exception, switches to supervisor state, and causes an immediate branch to a specific memory location, a.k.a. "vector"**
  - ***The operating system has code at that vector to save state and branch to a handler routine***

Notes: Exceptions

This page intentionally left blank.

Exceptions

- **Examples:**
  - ***system reset - vector 0x100***
  - ***machine Check - vector 0x200***
  - ***data storage interrupt (DSI) - vector 0x300***
  - ***instruction storage interrupt (ISI) - vector 0x400***
  - ***alignment - vector 0x600***
  - ***program (invalid instruction or trap instruction) - vector 0x700***
  - ***floating-point unavailable - vector 0x800***
  - ***decrementer - vector 0x900***
  - ***system call - vector 0xc00***
  - ***There are some exceptions unique to each type of processor.***

Notes: Exceptions

The system reset exception is used when a system reset is initiated by the system administrator. This generally causes a "soft" reboot of the system.

The machine check exception is generated when a hardware machine check occurs. This generally indicates either a hardware bus error or bad real address access. This exception can be disabled with the machine check enable (ME) bit in the machine state register (MSR). If a machine check occurs with the ME bit off, then a machine checkstop occurs. Generally, a machine check exception causes a kernel crash dump to be generated. A machine checkstop causes no kernel crash dump to be generated, though a checkstop record is generated.

Data storage interrupt (DSI) and instruction storage interrupt (ISI) exceptions are caused by hardware not being able to find a translation for a instruction fetch or load/store operation. These generally result in a page fault.

Alignment exceptions are generated when an instruction generates an unaligned memory operation that can not be completed by the hardware. Which unaligned operations cannot be handled by the hardware are processor dependent. This exception generally results in AIX performing the unaligned operation with special purpose code.

The program instruction is generated when an illegal instruction or trap instruction is generated. This is generally caused by debugger breakpoints in a process being hit. This exception generally results in a call to an application or kernel debugger.

The floating point unavailable exception is caused when a thread executes a floating point instruction when the machine state register (MSR) indicates that floating point operations are not allowed (FP bit not set). This generally indicates that a thread has not executed any floating point instructions yet or that another thread's floating point data is currently in the processor's floating point registers. AIX does not save a thread's floating point register values until it first uses the floating point registers. On UP systems, AIX does not save off floating point registers for the currently running thread when another thread is dispatched. Often, no other thread will use the floating point registers before the thread is again dispatched. This saves AIX having to save and restore the floating point registers on every thread dispatch.

The decrementer exception is caused when the decrementer register has reached the value zero. This indicates that a

Interrupts

- One special type of exception is driven by external hardware interrupts.
  - *external interrupt - 0x500*
- This exception allows asynchronous interruption of the processor by external devices.
  - *All I/O interrupts are routed through this exception.*
- Interrupts are guaranteed to occur between the execution of instructions.

Notes: Interrupts

On modern processors, many instructions are being executed at one time. When a hardware interrupt occurs, some set of those instructions are executed to completion and any following instructions are terminated with no effect on the processor registers or memory. This is what is meant by "interrupts are guaranteed to occur between the execution of instructions". The processor makes sure that the effect of its operations are equivalent to an interrupt occuring between the execution of instructions.

Interrupt Handling

- **When an interrupt is received, AIX must:**
  - ***save the current state of the machine,***
  - ***determine the real handler for the interrupt,***
  - ***call that handler to "service" the interrupt, and***
  - ***restore the machine state if and when the handler completes.***

Notes: Interrupt Handling

This page intentionally left blank.

Interrupt Priorities

- Iterrupt priorities have no relationship to process and thread scheduling priorities.
- AIX associates priorities with each type of interrupt.
  - *A lower priority number means a more favored interrupt.*
  - *The actual priorities numbers are covered later in this class.*
- Interrupt processing can itself be interrupted, but only by a more favored (lower priority number) interrupt.

Notes: Interrupt Priorities

This page intentionally left blank.

Interrupt Priorities

- **Programs running in kernel mode can manually raise their interrupt priority.**
  - ***This is called "disabling interrupts".***
- **Non-I/O exceptions are always delivered to AIX, regardless of the interrupt priority.**
  - ***AIX may decide to handle the exception, queue it as an interrupt, or raise an error.***

Notes: Interrupt Priorities

This page intentionally left blank.

Saving and Restoring Machine State

- **AIX maintains a set of *machine state save* (*mstsave*) areas.**
- **Each processor has a pointer to the mstsave area it should use when the next interrupt occurs.**
  - ***This pointer is called the* current save area, *or* csa pointer.**
- **When state needs to be saved, AIX:**
  - ***saves almost all registers into the mstsave pointed to by this processor's csa,***
  - ***gets the next available mstsave area from this processor's pool,***
  - ***links just-saved mstsave to new mstsave, and***
  - ***updates this processor's csa to point to new area.***

Notes: Saving and Restoring Machine State

Whenever AIX receives an interrupt that is of higher priority than what it is currently doing, it must save the state of the machine into an **mstsave** area. The **csa** (Current Save Area) pointer points to an unused mstsave area that AIX can use if another, higher-priority interrupt comes in. This area may contain stale data from being used for a previously-handled interrupt, but its prev pointer always points to the previous mstsave area (or is null if there aren't any more in use at that time).

These areas are linked together from most-recently to least-recently used, so this means that they go from higher to lower interrupt priority. At the end of the mstsave chain is the mstsave area for the base interrupt level. This mstsave area contains the state of the machine when it was last doing something other than interrupt processing (i.e. the machine state when the oldest interrupt that we are currently processing came in).

The stack used by an interrupt handler is kept in the same page with the mstsave area. This limits the stack to 4K minus the size of the mstsave area. Using this area for the stack ensures that the stack is pinned, which is required for interrupt handlers.

Saving and Restoring Machine State

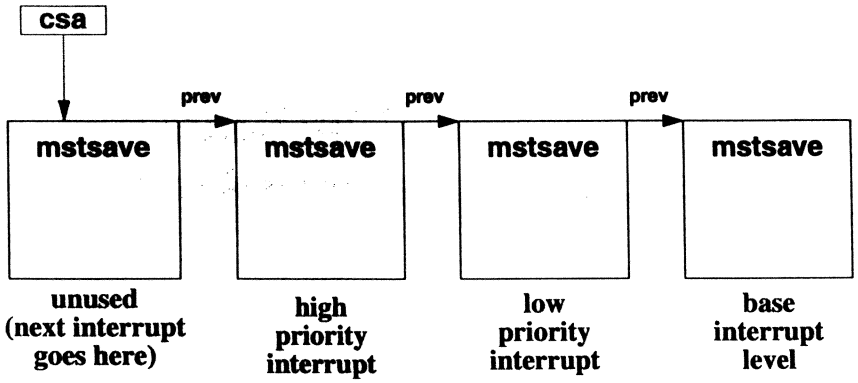
- **When an interrupt handler returns, AIX must restore the machine state that was in effect when the interrupt occurred.**
  - ***AIX does this by reloading registers from the processor's previous mst-save area.***
  - ***AIX then sets the processor's csa pointer to the (now unused) previous mstsave area.***
- **If returning to base interrupt level, AIX generally reruns the dispatcher to determine which thread to resume.**
  - ***The interrupt might have made another thread runnable.***

Notes: Saving and Restoring Machine State

This page intentionally left blank.

Saving and Restoring Machine State

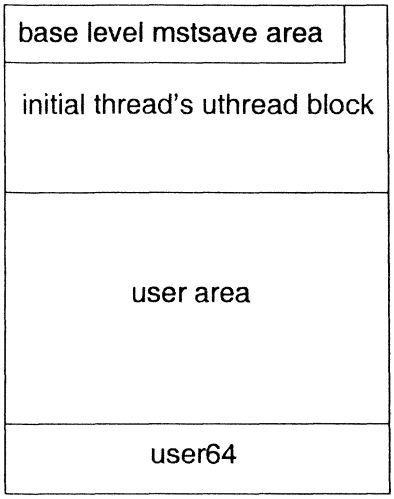
- At any point in time, mstsave areas provide an interrupt history stack:



Notes: Saving and Restoring Machine State

This page intentionally left blank.

Saving and Restoring Machine State



The initial thread's base level state save area is in the initial thread's uthread block.

The initial thread's uthread block is in the process' ublock

The user64 area is used for saving the 64-bit user registers for 64-bit processes

Notes: Saving and Restoring Machine State

The user64 area is only used created and pinned when the process is a 64-bit process. It is created when a process calls exec() for a 64-bit executable. It is destroyed when a 64-bit process exits or calls exec() for a 32-bit executable.

The portion of the base level state save area that contains the 32-bit registers is unused for 64-bit processes.

Only the base level state save area needs to have a 64-bit register state save area (user64) associated with it. Since the kernel only runs in 32-bit mode and all interrupt handlers run in kernel mode, all state save areas other than the base level state save areas need only save 32-bit state (even on 64-bit hardware).

Memory Management

Notes: Memory Management

Traditionally, the memory management component of the operating system is responsible for managing the system's real memory resources. Virtual memory systems provide the capability to run programs whose memory requirements exceed the system's real memory resources by allowing programs to execute when they are only partially resident in memory and by utilizing Direct Access Storage Device (DASD) to extend memory. The virtual memory system divides real memory into fixed-length pages and allocates pages to program as it requires them. Such a *demand-paging* system allows multiple programs to reside in memory and execute simultaneously. The virtual memory system is responsible for keeping track of which pages of a program are resident in memory and which are on secondary storage. It handles interrupts from the address translation hardware in the system to determine when pages must be retrieved from secondary storage and placed in real memory. When all of real memory is in use, it decides which program's pages are to be replaced and paged out to secondary storage.

Each time a process access a virtual address, the virtual address is mapped (if not already mapped) by the VMM to a physical address where the data is located. If the data is not in physical memory, it is loaded from disk or initialized. Loading a page from disk is called *pagein*. If there are no physical pages available to hold the data, a currently mapped physical page is used. If the page is unmodified, the page is just used for the new data. If the page was modified, the page is written to a paging device or file system and unmapped. This operation is called a *pageout*. The data from the disk requested is then loaded to the free physical page.

The VMM also provides for data sharing among programs and for access protection to prevent illegal access to data.

Outline

- Virtual Memory
- Page to Frame Mapping
  - *Page Fault Handling*
  - *Exceptions*
- Paging Space
- Address Spaces
- Segment Types
  - *File Mapping*
  - *Process Private Segments*

Notes: Outline

This page intentionally left blank.

Virtual Address Space

- The effective address space is the range of addresses that a process (or the kernel) is allowed to reference.
- The virtual address space is the set of all memory objects that could be made addressible by the hardware.
- The 32-bit effective address on AIX is broken into 16 segments, each 256MB in size. The total 32-bit effective address space is therefore 4 gigabytes.
- The 64-bit effective address on AIX is broken into 2^{36} segments, each 256MB in size. The total 64-bit effective address space is therefore 16 exabytes.
- Processes have access to a limited range of virtual addresses given to them by the kernel.

Notes: Virtual Address Space

This page intentionally left blank.

Virtual Address Space

- **The effective address space is mapped to different physical address space or disk files for each process.**
  - ***For example: two processes may have access to virtual address 0x30000000, but each process is referring to a different physical address or file object.***
  - ***A context switch causes the effective virtual mapping for the selected process to take effect.***
- **The physical address space is dependent on how much memory (memory chips) are on the machine.**
- **A machine can have a virtual address space that is larger than the physical address space on the machine.**
- **Physical address space maps one-to-one with the machine's hardware memory.**

Notes: Virtual Address Space

This page intentionally left blank.

Demand Paging

- Data is copied to a physical page only when referenced.
- Paging is done on the fly and is invisible to the user.
- Data comes from:
  - *A page from the page space.*
  - *A page from a file on disk.*
- In AIX, the kernel is pageable.
- Parts which cannot be paged, are pinned.
- Pager attempts to keep pool of physical pages free.

Notes: Demand Paging

Demand Paging:

AIX is a demand paging system. Physical pages (frames) are not allocated for virtual pages until they are needed (referenced). When a virtual address is referenced on a page that has no mapping to a frame, the mapping is done on the fly and the page frame is loaded from where it is mapped. The loading is invisible to the user process.

Demand paging saves much of the overhead of creating new processes because the pages for execution do not have to be loaded unless they are needed. If a process never uses parts of its virtual space, valuable physical memory will never be used.

Unlike most UNIX platforms, AIX's kernel is also pageable. Only those parts of the kernel referenced are in physical memory. Additionally, Kernel pages unused recently can also be paged out. Some parts of the kernel are required to stay in memory because it is not possible to perform a pagein when those pieces of code execute. If pages are required to stay in physical memory, the pages are said to be pinned. The bottom halves of devices drivers (interrupt processing) are pinned. Most of the kernel is not pinned.

A pager daemon attempts to keep a pool of physical pages free. If the number of pages available goes below a high-water mark threshold, the pager frees the oldest (referenced further back in time) pages until a low-water mark threshold is reached. This is the Least Recently Used (LRU) algorithm.

Memory Management Definitions

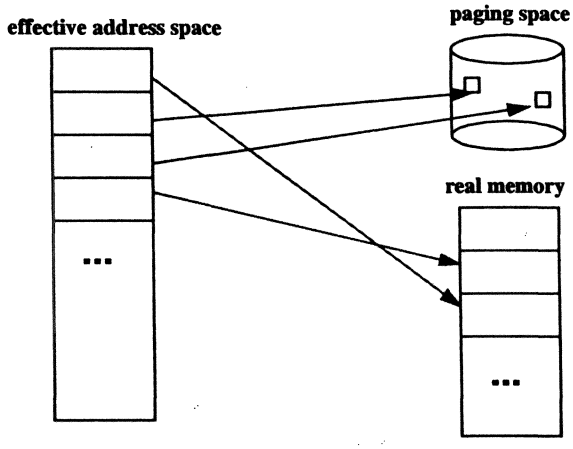
- **Page - a fixed size piece of memory**
  - ***Size = 4096 bytes for AIX***
- **Paging space - disk area(s) used by a memory manager to hold inactive memory pages with no other home**
- **Real memory - memory physically present inside a system unit**
  - ***e.g. SIMMs, memory cards, etc.***
- **Virtual memory - a picture of memory presented to programs/processes by a system memory manager**
- **Address space - the set of addresses available to a program that it can use to access memory**

Notes: Memory Management Definitions

This page intentionally left blank.

Effective Address Space

- Programs/processes see one contiguous address space
- Usually spread between real memory and paging space



Notes: Effective Address Space

This page intentionally left blank.

Virtual Memory

- The mapping of effective addresses to real addresses is generally done by hardware.
- The smallest unit of memory managed by hardware and software is 1 page.
- A *page fault* occurs when a program tries to access a page that is not currently in real memory.
  - *This causes a data storage interrupt or instruction storage interrupt exception.*
- The job of a virtual memory management system is to handle these page faults so that they are transparent to the thread using virtual memory addresses.

Notes: Virtual Memory

Virtual memory systems present to user programs the appearance of one large contiguous address space that they can use, even though the space presented may be larger than the physical memory available on the system. This is accomplished by only having memory that has been recently used in real memory, while memory that has not been recently used is kept aside in paging space.

For speed, most systems have the mapping of virtual addresses to real addresses done in the hardware. This mapping is done on a page-by-page basis. When the hardware finds that there is no mapping to real memory, it raises a page fault condition. The operating system software must handle these faults in such a way that the page fault is transparent to the user program. This is the key function of a virtual memory manager.

POWER & PowerPC Addresses

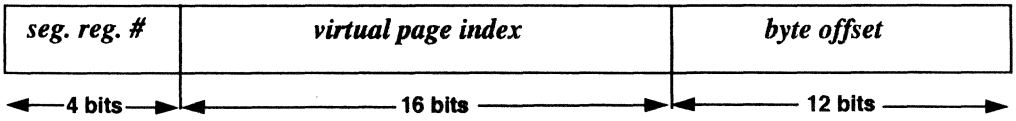
- **Addresses contain 32 bits on 32-bit processors and 64 bits on 64-bit processors.**
- **Effective addresses can be treated differently depending on whether they are instruction or data addresses.**
- **To determine whether an effective address is treated as a real or virtual address, the processor looks in the machine state register (MSR)**
  - ***If the address is generated to fetch an instruction, the MSR bit for IT (instruction translation) is examined. If it is on, the effective address is treated as a virtual address, otherwise as a real (physical) address.***
  - ***If the address is generated to load data, the MSR bit for DT (data translation) is examined. If it is on, the effective address is treated as a virtual address, otherwise as a real (physical) address.***

Notes: POWER & PowerPC Addresses

The hardware treats instruction addresses (where the processor goes to get an instruction) and data addresses (addresses generated by instructions to load or store data) differently. Whether an address is taken as real or virtual depends on the IT (instruction translation) or DT (data translation) bit in the Machine Status Register (MSR). The IT or DT bit being on indicates that the address is virtual (i.e. it needs to be translated). Otherwise, the address is taken to be a real address which is directly used to address the physical memory on the system.

Hardware Address Translation

- **Real addresses go straight to the physical memory, without translation.**
- **A 32-bit virtual address is broken into the following components:**



** The first 4 bits select the segment register, which contains the segment ID*
** The next 16 bits select the page within the segment*
** The next 12 bits select the offset within the page*

Notes: Hardware Address Translation

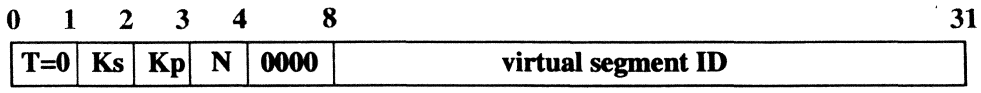
In a virtual address, the top 4 bits select a segment register, the next 16 bits select the location of the page within the segment and the last 12 bits select the offset within the page.

So...

- 2^4 = 16 (4 bits select one of 16 segment registers)
- 2^28 = 256 Meg (each segment is 256 Megabytes in size, 16 + 12 = 28)
- 2^12 = 4K (each page is 4K in size)
- 256M/Segment *16 segments = 4 Gigabytes address space per process
- 2^32 = 4 Gig (32 bit virtual address space)

Segments and Segment Registers

- A segment is a 256M (2^28) piece of the virtual address space.
- Segment registers select a specific segment to "fill in" a particular 256M piece of virtual address space.
- A 32-bit processor has 16 segment registers to represent a 32-bit address space.
- the 32-bit segment register format:



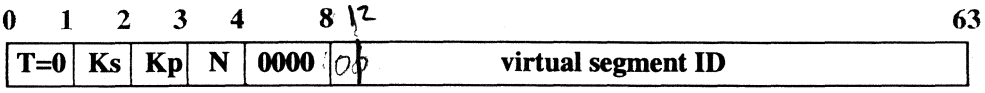
T=0 - always 0 for memory space
Ks - supervisor-state protection key
Kp - user-state protection key
N - no-execute protection

Notes: Segments and Segment Registers

This page intentionally left blank.

64-bit Segment Table

- A segment table is set-associative cache of 256 most recently used segment number to segment ID mappings.
- The address space register (ASR) points to the current segment table for the hardware.
- The segment lookaside buffer (SLB) caches segment number to segment ID mappings in the processor.
- the 64-bit segment table entry (STE) format:

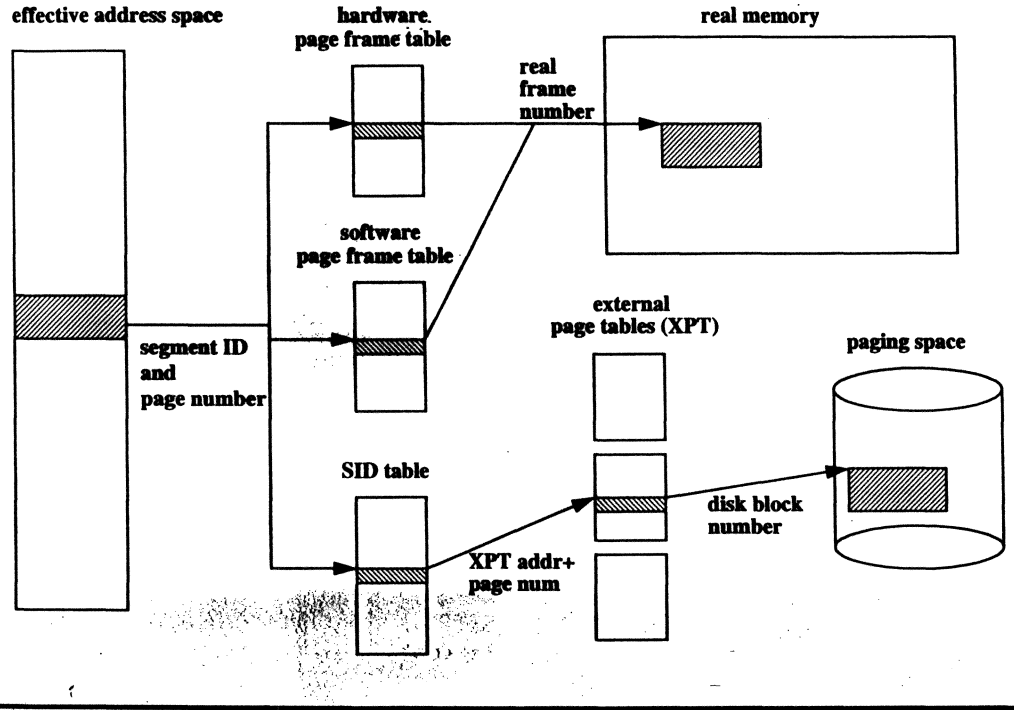


T=0 - always 0 for memory space
Ks - supervisor-state protection key
Kp - user-state protection key
N - no-execute protection

Notes: 64-bit Segment Table

This page intentionally left blank.

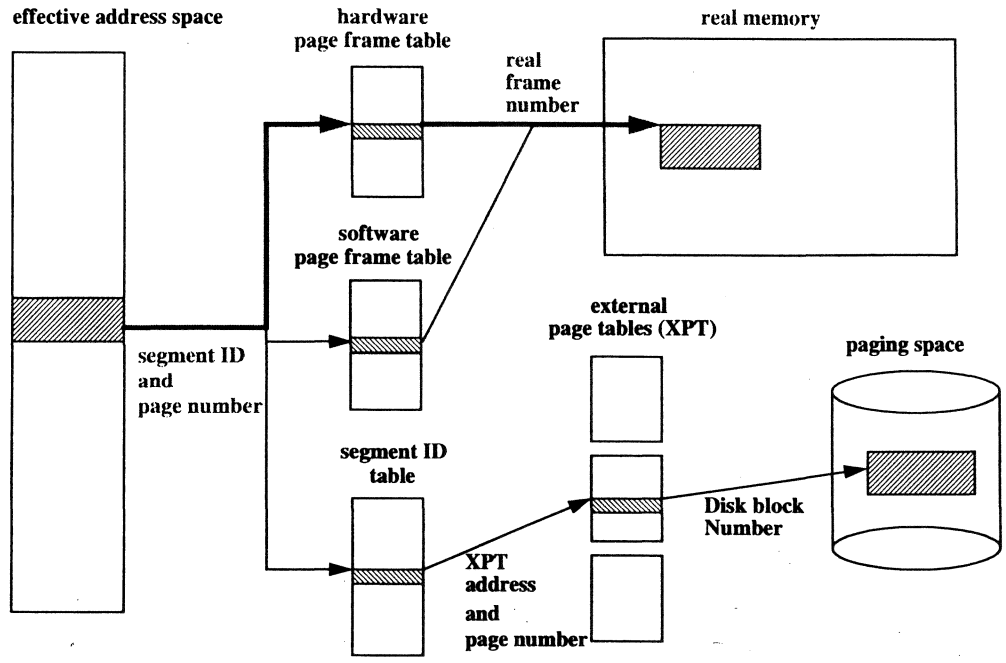
Page Mapping Overview



Notes: Page Mapping Overview

This diagram shows the overall relationship among the major AIX data structures involved in mapping a virtual page to a real page or to paging space. In the following sections, each of the major paths will be discussed in detail.

Hardware Page Mapping



Notes: Hardware Page Mapping

The processor generates a hash function value based upon the segment ID and segment page number of the page that was faulted upon. This hash function is determined by the processor architecture. The hash value generated is used to look up the appropriate hardware page frame table entries for the hardware to search for the correct page frame table entry.

If the hardware finds a hardware page frame table entry that has the correct segment ID and segment page number, it uses the real frame number to generate the real address for the page. The hardware does this by concatenating the real frame number with the page offset of the effective address.

Hardware Page Mapping

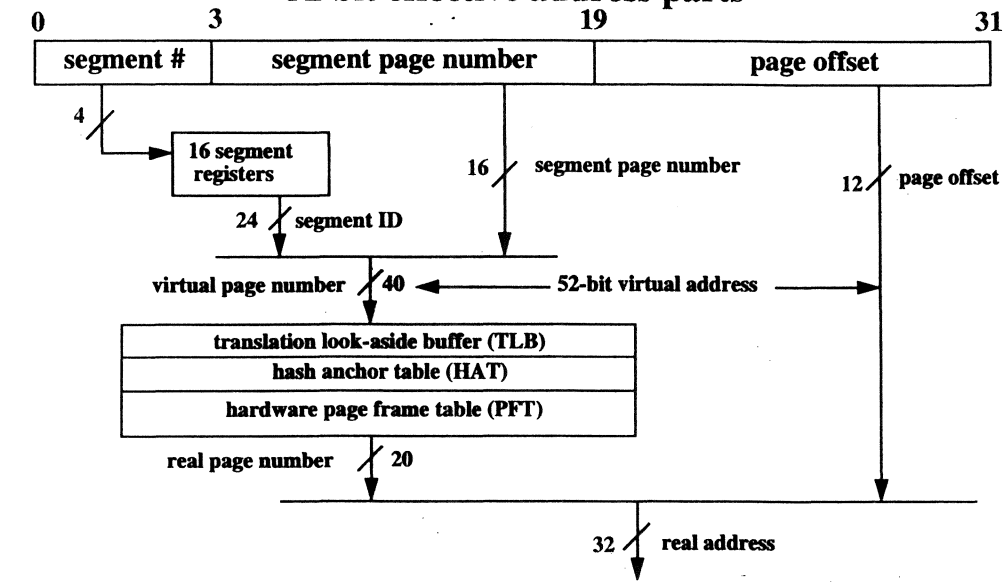
- **In the normal situation, an effective address refers to a piece of memory that is currently in real memory.**
  - ***We say the memory is already "paged in".***
  - ***The memory may also be "pinned" - marked so that it will always be in physical memory.***
- **The hardware knows everything necessary to locate the appropriate real memory location.**
  - ***AIX has already set up the information the hardware needs beforehand.***
  - ***AIX does not get involved in every memory access.***
- **The hardware page frame table (PFT) may not contain references for every page in real memory.**
  - ***Memory may have been "hidden" by AIX to prevent inadvertent access while the memory is in an inconsistent state (e.g. DMA).***

Notes: Hardware Page Mapping

In a properly-configured system running reasonable applications, referenced data will be in memory far more often than not. This allows the hardware to handle all address translation duties itself, leading to very fast access times. AIX does the work to set up the tables required by the hardware, then gets out of the way to let the hardware do its work.

The term "paged in" refers to virtual memory that is currently present in real memory (as opposed to being out on paging disk space). In some cases, it is critical to ensure that certain memory pages are **always** in real memory (or at least, for a fixed period of time). These pages are "pinned" via a kernel service that marks the pages as being ineligible to be taken out of real memory. One example of pinned memory is the set of all tables that the hardware needs to find virtual to real page mappings (e.g. the Hardware PFT).

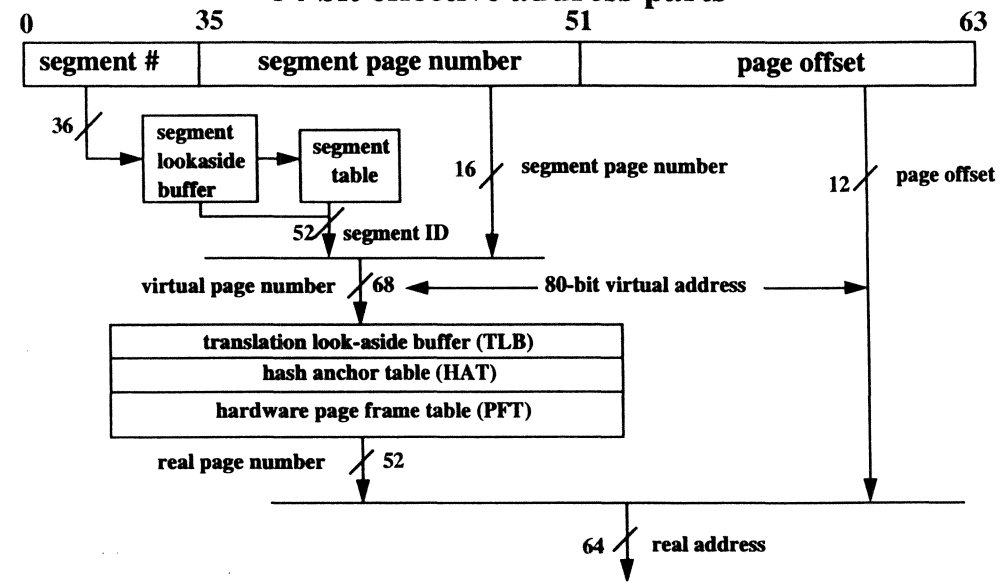
32-bit Hardware Address Resolution



Notes: 32-bit Hardware Address Resolution

POWER and PowerPC hardware allow the operating system to define a virtual memory space that is significantly larger than the maximum amount of real memory that can be addressed. On 32-bit hardware, this is accomplished via the use of segment registers. On 32-bit hardware, each 32 bit effective address uses the first 4 bits to select a segment register. The segment register contains a 24 bit segment ID. These 24 bits are used with the 16 bit segment page number from the original address to yield a 40 bit virtual page number. Combine this with the 12 bit page offset, and we get a 52 bit virtual address which is used internally by the processor. The 40 bit virtual page number is then used in a lookup mechanism to find a 20 bit real page number, which is combined with the 12 bit page offset to end up with a 32 bit real address.

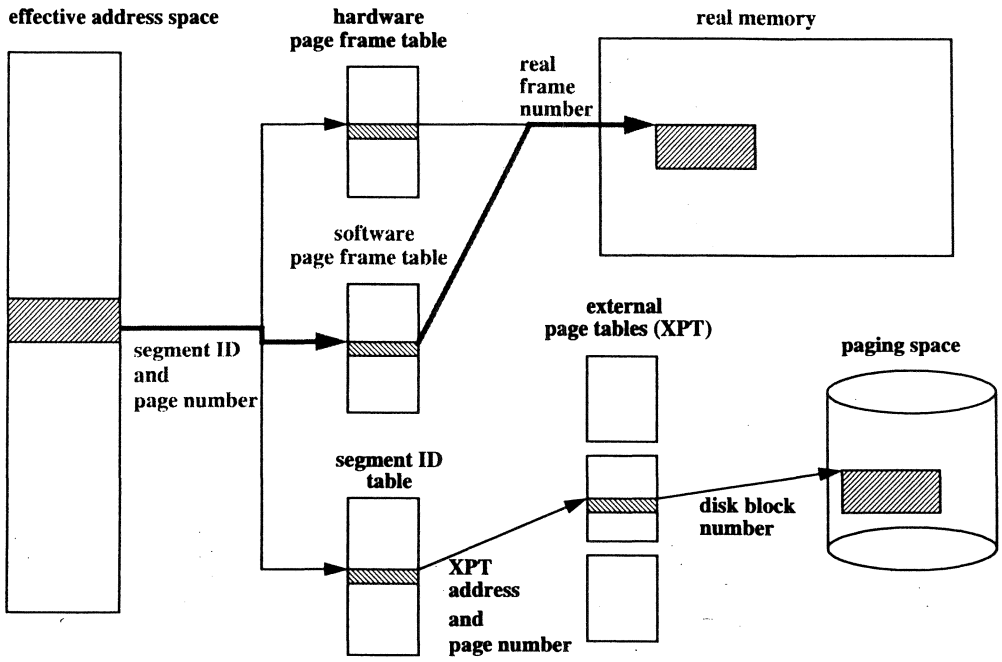
64-bit Hardware Address Resolution



Notes: 64-bit Hardware Address Resolution

64-bit PowerPC hardware allows the operating system to define a virtual memory space that is significantly larger than the maximum amount of real memory that can be addressed. This is accomplished via the use of a segment table. Each 64 bit effective address uses the first 36 bits as a segment number. The segment number is mapped to a 52 bit segment ID (using either a the segment lookaside buffer (SLB) or a segment table). These 52 bits are used with the segment page number from the original address to yield a 68 bit virtual page number. Combine this with the 12 bit page offset, and we get a 80 bit virtual address which is used internally by the processor. The 68 bit virtual page number is then used in a lookup mechanism to find a 52 bit real page number, which is combined with the 12 bit page offset to end up with a 64 bit real address.

Page Fault Handling - Page in Memory



Notes: Page Fault Handling - Page in Memory

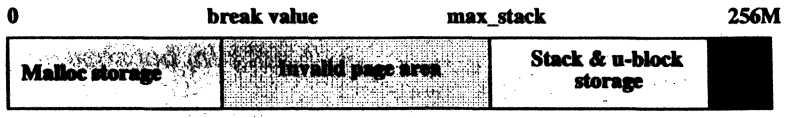
Page faults occur when the hardware has looked through all its tables but cannot find a real page mapping for the virtual page number it calculated from a virtual address. This is when the AIX Virtual Memory Manager (VMM) does the bulk of its work. It handles the interrupt by first verifying that the requested page is valid. To do this, it looks up the requested segment ID, verifies that it is a valid segment, then looks at the limit values for that segment (one for bottom-up allocations, and the other for top-down). If the page falls between the two limits, then it is invalid, and a kernel exception is generated.

If the page is valid, the VMM starts looking through the software PFT for the page. This processing almost duplicates the hardware processing, but it uses a different PFT. If the page is found, the VMM looks to see if the page is hidden (for example, there may be DMA scheduled for this page). If it is hidden, then the VMM tells the dispatcher to move the process or thread to the wait queue, suspending the process or thread until the page is no longer hidden. If the page is valid and is not in memory, it is loaded from paging space or the filesystem, the hardware PFT is updated, and the process/thread resumes at the faulting instruction without rerunning the dispatcher. The net effect is that the process or thread has no knowledge that a page fault occurred except for a delay in its processing.

Note: Even if we are not hiding a page, it may not be in the hardware PFT due to its size limits. PowerPC processors generate a hash value based upon the segment ID and segment page number to determine which page table equivalence groups (PTEGs) to search for the translation. These groups can only hold 16 page frame table entries each. Since there may be more than 16 pages that hash into one PTEG, the VMM has to decide which translation should be left in the PTEG. Then, when a page fault occurs for a page with a translation not in the PTEG, VMM only has to reload a PTEG entry with the translation for the page in question, replacing the translation for some other page. This is called a reload fault an may occur at any time, even for pinned pages. This requires the VMM to do this work without enabling interrupts.

Page Fault Handling - Page in Memory

- **When the hardware can't find a specified virtual page, it generates a page fault exception.**
  - ***either a data storage interrupt or instruction storage interrupt exceptions***
- **Virtual Memory Manager (VMM) takes the exception, then checks that the page is valid and page faults are allowed.**
  - ***Every segment has two limit values, bottom-up and top-down.***
    - ***Only process private segments uses the top-down limit.***
  - ***If the segment page number falls between the two limits, it is not valid.***
    - ***For example, if it's a process private segment, the segment page number must be below the break value or above max_stack:***



Notes: Page Fault Handling - Page in Memory

This page intentionally left blank.

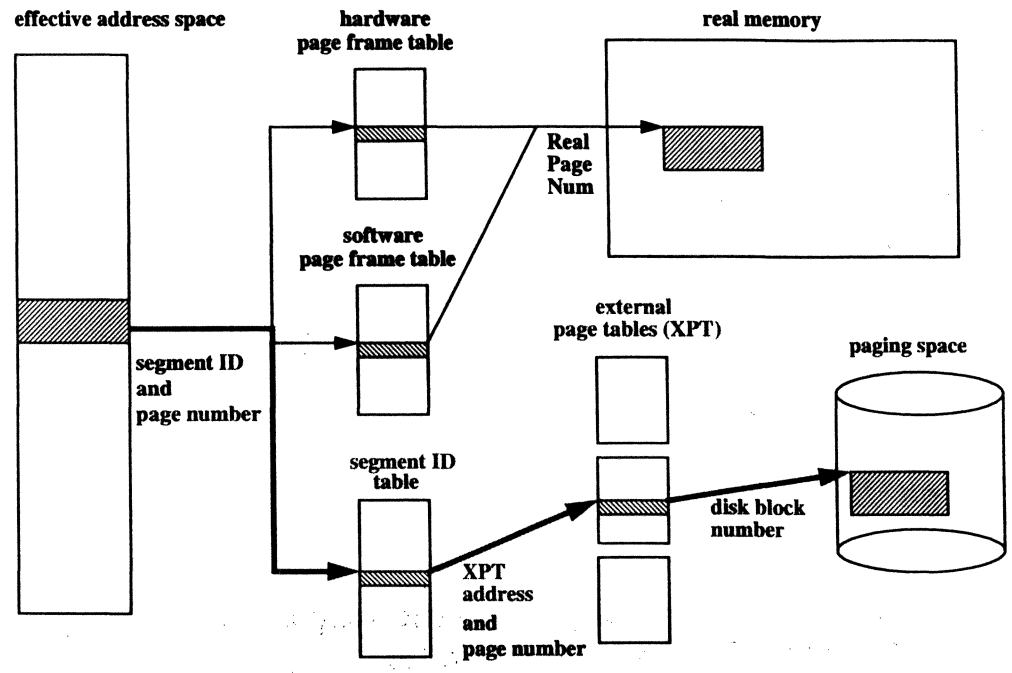
Page Fault Handling - Page in Memory

- **If the page is valid, VMM first looks in the software page frame table (PFT) for the page.**
  - ***The page might be in real memory after all, just not in the hardware PFT.***
  - ***Software PFT processing is similar to hardware PFT processing.***
    - ***Note: The software PFT itself is pinned.***
- **If page is found:**
  - ***If the page currently should be hidden, the thread requesting the page is suspended.***
  - ***Otherwise, the hardware PFT is updated with the real page number for this page and the process resumes execution.***
    - ***In this case, the dispatcher does not run, because no other thread is made runnable by this operation.***
  - ***No "page in" of the page occurs.***

Notes: Page Fault Handling - Page in Memory

This page intentionally left blank.

Page Fault Handling - Page on Disk



Notes: Page Fault Handling - Page on Disk

This page intentionally left blank.

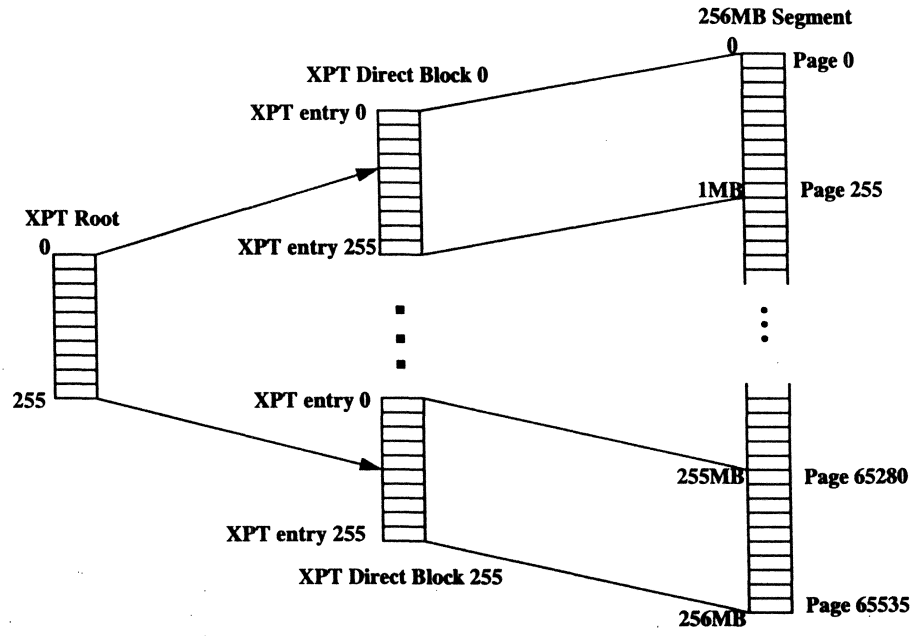
Page Fault Handling - Page on Disk

- If the page was not found in real memory, the VMM determines whether it is in paging space.
- The VMM looks up the segment ID for this address in the segment ID table and gets the external page table (XPT) root pointer.
  - *There is an XPT for all segments that are not associated with files.*
- The VMM finds the correct XPT direct block from XPT root.
- The VMM gets the paging space disk block number from XPT direct block.
- This mechanism is similar to the way inodes and indirect blocks are proce for file systems.

Notes: Page Fault Handling - Page on Disk

This page intentionally left blank.

External Page Table (XPT)



Notes: External Page Table (XPT)

This page intentionally left blank.

Page Faults - Loading a Page

- **If there is no XPT entry, then there is no paging space block allocated for this page.**
  - ***The VMM allocates a disk block from paging space and creates the XPT direct block (if necessary) and fills in the disk block number for the page.***
- **If there is a paging space block, load it into the page.**
  - ***1. VMM takes the first available frame from the free frame list (the free list contains one entry for each free frame of real memory).***
  - ***2. If the free frame list is empty, the VMM uses an algorithm to select several active pages to steal.***
    - ***If the page to be stolen is modified, an I/O request is issued to write the contents of the selected page to disk.***
    - ***Once written, the frames containing the stolen pages are added to the free list, and one is selected to hold the page from paging space.***
  - ***3. An I/O request loads the frame with the data for the faulting page.***

Notes: Page Faults - Loading a Page

Before the AIX 4.3.2 release, the VMM did not allocate a paging space disk block for a virtual memory page until the page was first referenced. This prevented long delays when memory was first allocated.

Starting in the AIX 4.3.2 release, the VMM waits to allocate paging space for a page until the page needs to be written out to paging space for the first time. This prevents long delays when memory is first allocated and when it is first used.

The VMM maintains a linked list containing all the currently free real memory pages in the system. When a page fault occurs, VMM just takes the first page from this list to assign to the faulting page. When the free frame list is empty and a page fault occurs, VMM selects several active pages to be stolen (usually around 20 or so), and all these pages are then added to the free list This reduces the amount of time spent starting and running the steal routines.

The mechanism used to determine which pages to steal is a pseudo-LRU (least recently used) algorithm called the "clock hand" algorithm. This algorithm is commonly used in operating systems when the hardware provides only a reference bit for each page in physical memory. The hardware automatically sets the reference bit for a page translation whenever a store occurs to the page. The clock hand algorithm checks frames by frame number looking for pages that have not been referenced since the last time the algorithm looked at the page. If a page has been referenced since the last time the algorithm looked at the frame, the algorithm clears the reference bit and goes to look at the next frame. If the page has not been referenced since the last time the algorithm looked at the frame, the page is stolen.

The algorithm is called the "clock hand" algorithm because the algorithm acts like a clock hand that is constantly pointing at frames in order. The "clock hand" advances whenever the algorithm advances to the next frame.

If a modified page is stolen, the clock hand algorithm writes the page to disk (to paging space or a file system) before stealing the page.

Note that copying a page from paging space to an available frame is not a synchronous process. Any process or thread waiting for a page fault to be handled is put to sleep until the page is available.

Page Faults Not Allowed

- **Page faults are not allowed if:**
  - ***an interrupt is being serviced, or***
  - ***interrupts are partially disabled.***
- **When interrupts are manually and completely disabled, page faults are still allowed.**
  - ***Faulting in this fashion should only be used by special kernel code.***
  - ***This mechanism is used for interrupt disabling code.***
  - ***This mechanism is used for locking code.***
- **If the VMM determines that page fault are not allowed, it causes a kernel exception.**

Notes: Page Faults Not Allowed

This page intentionally left blank.

Page Does Not Exist

- **A page does not exist if:**
  - ***the segment page number falls between the segment's limits, or***
  - ***segment ID invalid.***
- **A special invalid segment ID is used to indicate that there is no segment associated with that portion of the address space.**
- **If the VMM can't find the requested page, it causes a memory exception.**

Notes: Page Does Not Exist

This page intentionally left blank.

Segment Faults

- A segment fault occurs when a segment number of a 64-bit effective address is not in the current segment table on 64-bit hardware.
- The VMM searches the adspace B-tree for the current process looking for the segment number.
- If a segment ID mapping exists for the segment number, the segment table is loaded with the mapping and the instruction that caused the segment fault is restarted.
  - *This may require removing another segment number to segment ID mapping to make room for the new mapping.*
- If no segment mapping exists, this is treated like an invalid segment ID loaded into a segment register in a 32-bit address space.s

Notes: Segment Faults

This page intentionally left blank.

Memory Exceptions

- If we are in the kernel domain, any defined kernel exception handlers are invoked.
- If we are in the user domain, the user thread is sent a SIGSEGV signal.

Notes: Memory Exceptions

This page intentionally left blank.

Fatal Memory Exceptions

- **In all of the following cases, the VMM bypasses all kernel exception handlers and immediately halts the system:**
  - ***a page fault occurs in the interrupt environment***
  - ***a page fault occurs with interrupts partially disabled***
  - ***a protection fault occurs while in kernel mode on kernel data***
  - ***the system is out of paging space, or an I/O error occurs on kernel data***
  - ***an instruction storage exception occurs while in kernel mode***
  - ***a memory exception occurs while in kernel mode without an exception handler set up***

Notes: Fatal Memory Exceptions

This page intentionally left blank.

Segment Registers Revisited

- **The hardware definition of a segment register has 24 bits for use as a segment ID.**
- **The VMM uses 4 bits for its own purposes.**
  - ***Segment ID bit 0 (bit 8 in the segment register) is used in a hardware PFT entry to indicate that this page is scheduled for or undergoing I/O.***
    - ***This is used to hide the page from threads.***
    - ***The thread's segment register will not have the bit on, so the hardware won't find the translation and will generate a page fault exception.***
  - ***Segment ID bits 1-3 (bits 9-11 in the segment register) are used to allow the file system to access to 2Gb files without requiring the use of 8 segment IDs.***
- **So AIX only uses 48 bit internal virtual addresses.**
  - ***That still allows up to 256 terabytes of addressible objects.***

Notes: Segment Registers Revisited

This page intentionally left blank.

Memory Protection

- **Particular memory pages can be protected via a combination of segment register protection keys and page-protect (PP) bits.**
  - ***This page protection is provided by hardware.***
- **When the hardware is ready to load from or store to an address, it looks at the following items:**
  - ***the current execution state (user or kernel mode) in the machine state register (MSR)***
  - ***the key bit corresponding to execution state in segment register (Kp or Ks)***
  - ***the PP bits in hardware page frame table entry for the page***

Notes: Memory Protection

This page intentionally left blank.

Memory Protection

- **These are the combinations of segment register key bits and hardware page frame table PP bits that allow certain types of access:**

key	PP	allowed accesses
0	00	read/write
0	01	read/write
0	10	read/write
0	11	read
1	00	none
1	01	read
1	10	read/write
1	11	read

Notes: Memory Protection

This page intentionally left blank.

Memory Segments

- **All segments are broken into pages.**
  - ***Segments can be shared among processes.***
  - ***Segments can grow by adding additional pages.***
  - ***Segments can be attached or detached from processes.***
  - ***New segments can be created or destroyed by threads in a process.***

Notes: Memory Segments

This page intentionally left blank.

Memory Segments

- **types of segments:**
  - *working - temporary segment used during the execution of a program for its stack or data areas*
  - *persistent - segment used to hold file data for the local filesystems*
  - *client - segment used for pages of client filesystems (all filesystem types other than JFS)*
  - *log - segment used for writing or reading JFS filesystem logs during journalling operations*
  - *mapping - used to support the mmap() interfaces which allows an application to map multiple objects to the same memory segment*

Notes: Memory Segments

This page intentionally left blank.

Process Address Space

- **The 32-bit process address space is a 32-bit address space in user mode.**
  - ***The segments used are selected by the segment IDs in the process' segment registers.***
- **The 64-bit process address space is a 64-bit address space in user mode.**
  - ***The segments used are selected by the segment IDs in the process' segment table.***
  - ***The process address space is represented by the adspace structures.***
- **The kernel address space is a 32-bit address space.**
  - ***This is a subset of the 52-bit virtual address space on 32-bit hardware.***
  - ***This is a subset of the 80-bit virtual address space on 64-bit hardware.***

Notes: Process Address Space

For 32-bit processes on 64-bit hardware, the address space is represented as if it was in real segment registers. The 64-bit hardware acts as if it had 32-bit segment registers (segment register load and store instructions work), but they are really implemented differently. (The support for segment registers on 64-bit hardware is part of the "bridge extensions" IBM added to the PowerPC 64-bit architecture.)

The adspace structures for representing the 64-bit user address space are a B-tree representation of the large potentially sparse 64-bit address space. This representation results in not needing storage for the large empty regions of the 64-bit address space.

Process Address Space

- **The kernel loads several segment registers in a conventional way for all processes when they are created**
  - ***This provides the basic memory addressability needed by most processes.***
- **The segments are:**
  - ***kernel segment - (read-only) shared by all processes***
  - ***one or more segments for the main program of the current process***
    - ***These are shared on a read-only basis with other processes executing the same program.***
      - ***These generally map directly to the executable program in a filesystem.***
  - ***process private segment - (read-write) private to process***
  - ***shared library text segment - (read-only) shared by all processes***
  - ***shared library data segment - (read-write) private to process***

Notes: Process Address Space

This page intentionally left blank.

Process Address Space

- **Only the kernel can load a segment register.**
  - ***This enables the kernel to enforce access permissions for objects in virtual memory.***
- **Threads can modify their address space via system calls that change the process' segment registers.**
  - ***This can be used to map a file into memory.***
  - ***This can be used to attach to a shared memory segment.***
- **The kernel checks access permission in the system call.**
- **If access is allowed, the kernel gives the process access to the object by loading the object's segment ID into an unused segment register.**

Notes: Process Address Space

The instruction to load a segment register is a priveleged instruction. This means that the instruction can only be executed when the processor is running in kernel mode. If user mode code tried to execute this instruction, an illegal instruction exception would be generated by the hardware.

kernel mode effective address space (before AIX 4.3.2)

segment number	segment usage
0	kernel text and data (kernel segment)
1	file system data and message buffers segment
2	per-process data (process private segment)
3 through 10	available for temporary attaches
11	VMM data segment
12	VMM page frame table segment
13	VMM temporary usage segment
14	system process and thread data (kernel extension segment)
15	per-thread kernel data (uthread blocks and thread stacks)

Notes: kernel mode effective address space (before AIX 4.3.2)

The segment for kernel thread stacks and uthread blocks is only allocated and attached for multithreaded processes. The initial thread's stack and uthread block are in the process private segment, and for a single threaded process, this is all that is necessary.

There are potentially many kernel thread stack and uthread block segments for a single process. If the number of threads in the process is above a certain number (about 2,000), then there will be multiple of these segments for the process. The segment that will be attached will be the one that contains the kernel stack and uthread block for the currently running thread.

kernel mode effective address space (AIX 4.3.2)

segment number	segment usage
0	kernel text and data (first kernel segment)
1	file system data segment
2	per-process data (process private segment)
3 and 4	available for temporary attaches
5	kernel data (second kernel segment)
6	available for temporary attaches
7 through 10	message buffers
11	available for temporary attaches
12	VMM page frame table segment
13	VMM temporary usage segment
14	system process and thread data (kernel extension segment)
15	per-thread kernel data (uthread blocks and thread stacks)

Notes: kernel mode effective address space (AIX 4.3.2)

The space for message buffers in the AIX kernel was dramatically increased in the AIX 4.3.2 release (from 128 megabytes to 1 gigabyte). In this release, the amount of kernel data space was also increased from roughly 250 megabytes to over 500 megabytes. These changes caused significant reorganization of the kernel effective address space.

interrupt environment effective address space (before AIX 4.3.2)

segment number	segment usage
0	kernel text and data (first kernel segment)
1	message buffers segment
2	system use only (process private segment)
3 to 4	available for temporary attaches
5	kernel data (second kernel segment)
6 through 10	available for temporary attaches
11	VMM data segment
12	VMM page frame table segment
13	VMM temporary usage segment
14	system process and thread data (kernel extension segment)
15	system use only (uthread blocks and thread stacks)

Notes: interrupt environment effective address space (before AIX 4.3.2)

The process private segment is not supposed to be used by interrupt handlers. This is because interrupt handlers are not supposed to care about or affect the currently running thread. For the same reason, the uthread blocks and thread stacks should not be used by an interrupt handler.

The segment containing the kernel uthread blocks in segment register 15 is the same segment as the kernel thread stacks and uthread blocks in segment register 15 when not at interrupt level. Only the uthread blocks are accessible because they are pinned. The thread stacks are not pinned because they are not used at interrupt level. The interrupt level stack used (for all threads) is the one in the mst blocks.

interrupt environment effective address space (AIX 4.3.2)

segment number	segment usage
0	kernel text and data (first kernel segment)
1	reserved for system use
2	system use only (process private segment)
3 and 4	available for temporary attaches
5	kernel data (second kernel segment)
6	available for temporary attaches
7 through 10	message buffers
11	available for temporary attaches
12	VMM page frame table segment
13	VMM temporary usage segment
14	system process and thread data (kernel extension segment)
15	system use only (uthread blocks and thread stacks)

Notes: interrupt environment effective address space (AIX 4.3.2)

This page intentionally left blank.

32-bit user mode effective address space

segment number	segment usage
0	system call tables and low level routines (first kernel segment)
1	application text (application's file segment)
2	user data and user stack (process private segment)
3 through 10	shmat, mmap, or big data heap
11 and 12	shmat or mmap use
13	shared library text segment
14	shmat or mmap use
15	per-process shared library data segment

Notes: 32-bit user mode effective address space

The portions of the kernel segment that are visible in user mode are the SVC tables and the system call initial code. The SVC tables must always be allocated from the kernel heap in segment zero so that they can be mapped into the user address space.

Segments 3 through 0xA are available for user data and heap for "big data" programs. These programs have specified a **maxdata** value in their executable that indicates how big a data/heap region to allocate. For these programs, the initialized data and BSS for the executable is in segment 3 with the heap following. shmat and mmap for these programs can only occur beyond the range specified by the **maxdata** value. The text, initialized data, and BSS for loaded modules will still be in segment 2 (as they are for programs that are not "big data" programs).

64-bit user mode effective address space

segment number	segment usage
0	system call tables (first kernel segment)
1	reserved for system use
2	reserved for user mode loader (process private segment)
3 through 12	shmat or mmap use
13	reserved for user mode loader
14	shmat or mmap use
15	reserved for user mode loader
16 to end of text	application text
end of text to end of data	application initialized global data
end of data to end of BSS	application uninitialized global data (BSS)
end of BSS to end of heap	application heap
end of heap to 0x6FFFFFFF	available for application shmat and mmap
0x6FFFFFFF to 0x7FFFFFFF	default application shmat and mmap area
0x7FFFFFFF to 0x8FFFFFFF	application explicit module load area
0x8FFFFFFF to 0x9FFFFFFF	shared library text and per-process shared library data
0x9FFFFFFF to 0xEFFFFFFF	reserved for future system use
0xEFFFFFFF to 0xFFFFFFFF	application primary thread stack

Notes: 64-bit user mode effective address space

Segment numbers 3 through 12 are reserved for shmat and mmap use in 64-bit processes so that they can map memory that is shared with 32-bit processes at the same effective addresses. This allows embedded pointers in data structures to refer to other data in the shared memory region.

The portions of the kernel segment that are visible in user mode are the SVC tables and the system call initial code. The SVC tables for 64-bit processes must be allocated from the kernel heap in segment zero so that they can be mapped into the 64-bit user address space.

The **"maxdata"** attribute of an executable is ignored for 64-bit executables. This is because the normal data space for 64-bit programs can accommodate any size **maxdata** executable.

Memory Segments

- There are six important types of segments:
  - *kernel*
  - *user text*
  - *shared library text*
  - *shared data*
  - *process private*
  - *shared library data*

Notes: Memory Segments

This page intentionally left blank.

Kernel Segments

- **Kernel segments contain much of the global state of the kernel.**
- **The kernel has three segments that contain the global state of the operating system:**
  - ***in segment register 0, the first kernel segment***
  - ***in segment register 5, the second kernel segment***
  - ***in segment register 14, the kernel extension segment***

Notes: Kernel Segments

This page intentionally left blank.

Kernel Segments

- **The first kernel segment is divided into three parts:**
  - ***The first part contains the text and data of the base kernel module***
  - ***The second part is the kernel and pinned heaps***
    - ***Memory in the kernel heap may be pinned or unpinned***
    - ***Take up most of the rest of the first kernel segment***
  - ***The third part contains data structures for the kernel and pinned heaps***
    - ***These are statically set to very large sizes so that they can accomodate a wide range of machines and user configurations.***
    - ***These structure are pageable thus real memory is not wasted.***

Notes: Kernel Segments

This page intentionally left blank.

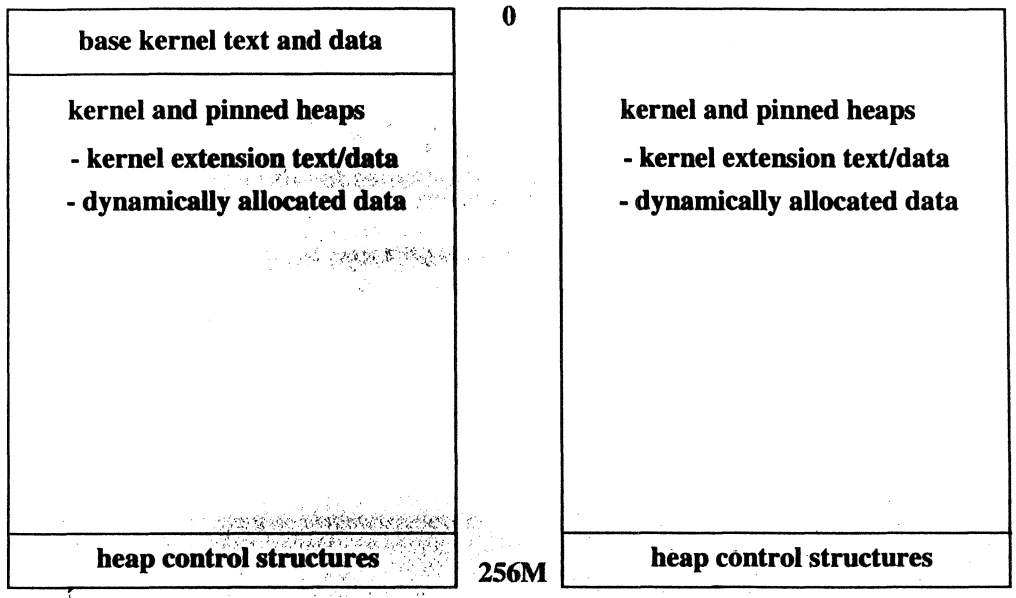
Kernel Segments

- **The second kernel segment is divided into two parts:**
  - ***The first part is the kernel and pinned heaps***
    - ***Memory in the kernel heap may be pinned or unpinned***
    - ***Take up most of the second kernel segment***
  - ***The second part contains data structures for the kernel and pinned heaps***
    - ***These are statically set to very large sizes so that they can accomodate a wide range of machines and user configurations.***
    - ***These structure are pageable thus real memory is not wasted.***

Notes: Kernel Segments

This page intentionally left blank.

Kernel Segments



Notes: Kernel Segments

Prior to the AIX 4.3.2 release, there was one kernel segment.

Starting in the AIX 4.3.2 release, there are two kernel segments. The kernel and its initialized and uninitialized global data are kept only in the first kernel segment, which is normally attached at segment register zero in kernel mode. The other kernel segment is used only for dynamic data allocation (the kernel and pinned heaps). The space remaining in the first kernel segment is also used for dynamic data allocation.

Kernel Extension Segment

- **The kernel extension segment has three parts:**
  - ***the process table (proc structures)***
    - ***starts at effective address 0xe3000000***
  - ***the thread table (thread structures)***
    - ***starts at effective address 0xe6000000***
  - ***the lock statistics (if lock instrumentation is turned on)***

Notes: Kernel Extension Segment

This page intentionally left blank.

User Text Segment

- **This segment contains program text.**
  - ***The segment is the same segment as for the application file.***
- **Threads in user mode have read-only access.**
  - ***Allows a single copy to be shared by all processes associated with the same program.***
- **one exception:**
  - ***A private read/write copy of the user text segment is provided for each process that is being debugged.***
    - ***This allows debuggers to set breakpoints directly in code.***
- **64-bit processes can have multiple segments to represent executable programs that are larger in size than 256 megabytes.**

Notes: User Text Segment

This page intentionally left blank.

Shared Library Text Segment

- Contains a copy of the program text (instructions) for the shared libraries currently in use in the system.
  - *Only one segment is used for 32-bit shared libraries. Many segments can be used for 64-bit shared libraries.*
- These segments are added to the user address space by the loader when the first shared library is loaded.
- Each process using text from this segment has a copy of the corresponding data in the per-process shared library data segment.
- Text for shared libraries is added to the segment starting at the lowest address.
  - *The space used grows as shared libraries are loaded for the first time.*

Notes: Shared Text Segment

This page intentionally left blank.

Shared Library Text Segment

- **Shared library text is loaded into this segment when an module is loaded via the *exec()* or *load()* system calls.**
  - ***Executable modules list the shared libraries they need at exec() time.***
  - ***Programs may issue load() calls to get additional shared modules.***
- **Any module loaded by a user process will be put in the shared text segment if:**
  - ***The permissions on the shared module allow read and execute access by others.***
  - ***There is room in the segment for the module.***
- **Otherwise, the module and its data are loaded into the user data area.**

Notes: Shared Library Text Segment

This page intentionally left blank.

Shared Library Text Segment

- User mode processes have read-only access.
  - *Otherwise, the segment couldn't be shared with other processes.*
- The same exception exists as for the user text segment.
  - *A private read/write copy of the shared text segment is provided for each process that is being debugged.*
- The shared library text segment is a global segment.
  - *Every process using shared text has the same segment attached.*

Notes: Shared Library Text Segment

This page intentionally left blank.

Shared Memory Segments

- A process can create and/or attach a shared data segment that is accessible by other processes.
- A shared data segment can represent a single memory object or a collection of memory objects.
- There are two kinds of memory objects, ATT System V memory objects, and mmap objects.
- System V memory object manipulation routines:
  - *shmget() - creates a shared memory object*
  - *shmat() - attaches a shared memory object*
  - *shmdt() - detaches a shared memory object*
  - *shmctl() - sets or gets the size and permissions of a shared memory object, or destroys a shared memory object*

Notes: Shared Memory Segments

This page intentionally left blank.

Shared Memory Segments

- mmap memory object manipulation routines:
  - *mmap() - creates and attaches an mmap memory object*
  - *munmap() - destroys an mmap memory object*
- The shared memory can be attached at different effective addresses in different processes.
- Shared memory can be attached read-only or read-write.
- There is no implicit serialization support when two or more processes access the same shared data segment
  - *Processes must coordinate their usage via semaphores or other IPC mechanisms.*

Notes: Shared Memory Segments

This page intentionally left blank.

Memory Mapped Files

- **Shared segments can be used to map any ordinary file directly into memory.**
  - ***Instead of reading and writing to the file, the program would just load or store in the segment.***
  - ***This avoids buffering of the I/O data in the kernel.***
  - ***This provides easy random access, as the file data is always available.***
  - ***This avoids the system call overhead of read() and write().***
- **Either the shmat() or mmap() system calls can be used.**
  - ***These services use the file segment as a shared memory object.***

Notes: Memory Mapped Files

The system allows file mapping at the user level. This allows a program to access file data through loads and stores to its virtual address space. This single level store approach can also greatly improve performance by creating a form of Direct Memory Access (DMA) file access. Instead of buffering the data in the kernel and copying the data from kernel to user, the file data is mapped directly into the user's address space. The file can even be shared between multiple processes even if some are using mapping and others are using the read/write system call interface. Of course, this may require some sort of synchronization scheme between the processes.

Another interesting aspect of the explicitly mapped file support is the ability to specify deferred update. With deferred update, the application can begin modifying the file data (by memory mapped loads and stores) and then either commit the modifications to the file system (via fsync()) or discard the modifications completely. This can greatly simplify error recover and allows the application to avoid a costly temporary file that may otherwise be required.

Memory Mapped Files

- ***shmat*() can be used to memory map a file.**
  - ***An open file descriptor is used in place of a shared memory ID.***
  - ***Once the file segment is mapped, it is treated like any other shared segment.***
    - ***It can be shared with other processes.***
- **Files can be mapped with one of three accesses allowed:**
  - ***read-write***
  - ***read-only***
  - ***deferred-update***

Notes: Memory Mapped Files

This page intentionally left blank.

Memory Mapped Files

- **read-write mapping:**
  - ***This allows loads and stores in the segment to behave like reads and writes to the corresponding file.***
  - ***If a thread loads beyond the end of the file, the load will load zero values.***
  - ***If the process stores beyond the end of the file, the file is extended to the following page boundary.***
- **read-only mapping:**
  - ***This allows only loads from the segment.***
  - ***Any attempt to store into the segment will generate a signal (SIGSEGV) to the thread.***
  - ***Just as with read-write access, a thread that loads beyond the end of the file loads zero values.***

Notes: Memory Mapped Files

This page intentionally left blank.

Memory Mapped Files

- **deferred-update mapping:**
  - ***This also allows loads and stores to the segment to behave like reads and writes to the corresponding file.***
  - ***The difference between this mapping and read-write mapping is that the modifications are delayed.***
    - ***Any storing into the segment modifies the segment but does not modify the corresponding file.***
- ***The fsync()* system call causes modifications to go to the corresponding file.**
  - ***This writes the changed portions of the segment to the file.***
    - ***This makes the mapped file an exact copy of the segment.***
  - ***If the fsync() system call is not issued, the file is never modified.***
    - ***This allows a program to cancel changes that it has "made to the file".***

Notes: Memory Mapped Files

This page intentionally left blank.

Memory Mapped Files

- **AIX exploits the close relationship between VMM and file systems to handle memory mapped files.**
  - ***File blocks become backing store for segments that map files.***
    - ***Instead of paging space, the file system blocks are used.***
  - ***Read-only mappings are handled using key bits and PP bits to protect the pages from store operations.***
  - ***Deferred-update is done by using a separate copy-on-write segment (not the file segment) and making the original pages read-only.***
    - ***When a protection page fault occurs, the VMM creates a new page backed by paging space.***
    - ***fsync() causes the VMM to copy the segment pages to the corresponding file segment pages.***
    - ***If no fsync() occurs, paging-space pages go away when process does.***

Notes: Memory Mapped Files

This page intentionally left blank.

Process Private Segment

- The process private segment contains:
  - *user data (for 32-bit programs that aren't* maxdata *programs)*
  - *the user stack (for 32-bit programs)*
  - *text and data from explicitly loaded modules (for 32-bit programs)*
  - *kernel per-process data (accessible only in kernel mode)*
  - *primary kernel thread stack (accessible only in kernel mode)*
  - *per-process loader data (accessible only in kernel mode)*
- It is not shared with other processes.

Notes: Process Private Segment

This page intentionally left blank.

Process Private Segment

- **32-bit program user data:**
  - ***By default, the data starts in the low addresses of the process private segment. "Big data" (or "maxdata") programs' data starts in segment 3.***
  - ***The process has read-write access to the user data in user mode.***
  - ***The size of the area can be changed using the brk() or sbrk() system calls.***
- **User data consists of:**
  - ***the initialized data and BSS sections of the program***
  - ***any memory dynamically allocated via* malloc*() or other similar subroutines (the application heap)***
  - ***shared library routines and data (if not in the shared library segments)***
- **Privately loaded routines and data for 32-bit programs are always allocated in the process private segment.**

Notes: Process Private Segment

Under some circumstances for 32-bit programs, shared library text and data won't be put in the shared library text and data segments. This is done when the text or data won't fit or does not have global read and execute permission. In these cases, the text and data will be put in the process private segment.

Process Private Segment

- **the user stack area:**
  - ***The stack starts at the high addresses in the process private segment for 32-bit processes and in the last stack segment for 64-bit processes.***
  - ***The stack grows in size toward smaller effective addresses as needed.***
  - ***The process has read-write access to the user stack in user mode.***
- **The size of the user stack and user data area are limited by the kernel.**
  - ***Limits can be queried or changed on a process basis via the getrlimit()/ setlrimit() system calls.***

Notes: Process Private Segment

This page intentionally left blank.

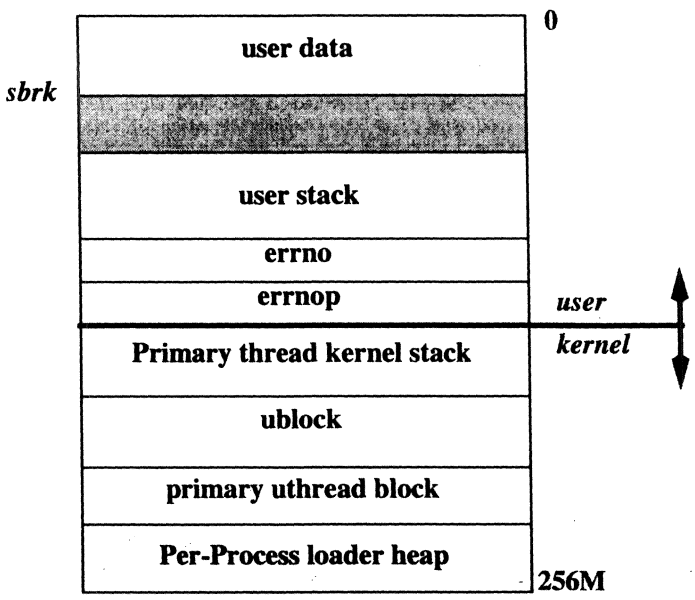
Process Private Segment

- **Per-process kernel data is kept above the user stack**
  - ***This data is not accessible in user mode.***
  - ***The per-process kernel data contains:***
    - ***the user structure and the primary uthread structures (the u-block)***
    - ***the kernel mode primary thread stack***
    - ***loader control structures (in the per-process loader heap)***
    - ***a heap structure (describing the per-process loader heap)***

Notes: Process Private Segment

This page intentionally left blank.

Process Private Segment Layout



Notes: Process Private Segment Layout

This page intentionally left blank.

Large Address Space Model

- **AIX supports a user data area larger than 256 megabytes for 32-bit processes.**
  - ***This is specified via the* maxdata *binder option.***
- **Large user data programs' data starts in segment 3.**
  - ***The heap can use up to 8 segments.***
    - ***Segments 11 and 12 can't be used for the user heap.***
  - ***The application heap starts after the application initialized data and BSS.***
- **Everything else stays in process private segment.**
  - ***loaded modules, shared library data, stack, etc.***
- **This model prevents these segments from being used for shared memory or file mapping.**

Notes: Large Address-Space Model

The large address-space model can be used for 32-bit user applications that need more than 256MB of dynamic data space. In the large address-space model, dynamically allocated user data starts in segment 3 and can take up to eight segments. Therefore, the user application has 2GB of data space available (eight times 256 megabytes).

Note: The user text size and initialized data are still limited to 256MB in segment 1 and the user stack has something less than 256MB available in segment 2. Using segments for data space means that those segments used are unavailable for mapping with the mmap() or shmat() system calls.

To use the large address-space model, the maxdata flag specifies the number of additional segments to cc or ld that the application will use. The maxdata flag sets the o_maxdata attribute used by the loader. For example:

```
cc -o sample -bmaxdata:0x50000000 sample.c
```

specifies that the sample module will use up to five segments. However, up to eight segments can be used. The maxdata flag should only be set if the large address-space model is needed, as using this model is less efficient.

When the large address-space model is used, system calls such as malloc, sbrk, and so on, will use segment 3-10 to allocate space instead of segment 2. Segment 11 and 12 cannot be used by the large address-space model.

Note: Programs that use large amounts of data space will require equally large amounts of paging space if the data pages are actually referenced or written into. If paging space runs low on the system, the offending process or other processes may be terminated.

Per-Process Shared Library Data Segment

- This segment holds data items required by modules in the shared text segment(s).
- There is one of these segments for each process.
  - *This segment acts like an extension of the process private segment.*
- Addresses of data items generally the same across processes
  - *So shared text modules can refer to most data items without worrying about process-specific relocation*
- Data itself is not shared
- These segments start as copies of the System Shared Library Data Segment

Notes: Per-Process Shared Library Data Segment

This page intentionally left blank.

Process Management

Notes: Process Management

This page intentionally left blank.

Outline

- Process Management
  - *Process States*
- Kernel Processes
- Processes and Threads
  - *Thread States*
- Thread Scheduling
- Process and Thread Data Structures
- M:N Threads
- Signals

Notes: Outline

This page intentionally left blank.

Process Management

- A process consists of:
  - *A process table entry*
  - *A process ID (PID)*
  - *Virtual address space*
    - *User-area (U-area)*
    - *Program "text"*
    - *data*
    - *User and kernel stacks*
  - *Statistical Information*

Notes: Process Management

This page intentionally left blank.

Definition of Process Management

- Process management allows many processes to exist simultaneously and to share usage of the CPU.
- A process is a self-contained entity that consists of the information required to run a single program, such as a user application.
- The kernel contains a table entry for each process called the *proc* entry.

Notes: Definition of Process Management

This page intentionally left blank.

Definition of Process Management

- The *proc* entry contains information necessary to keep track of the current state and location of page tables for the process.
- The *proc* entry resides in a slot in an array of proc entries.
- The kernel is configured with a fixed number of slots.
- All processes have a process ID or PID.
- The PID is assigned when the process is created and provides a convenient way for users to refer to the other processes.

Notes: Definition of Process Management

This page intentionally left blank.

Definition of Process Management

- The process contains a list of virtual memory addresses that the process is allowed to access.
- The virtual memory is controlled by memory management.
- The user-area (*u_area*) of a process contains additional information about the process when it is running.

Notes: Definition of Process Management

This page intentionally left blank.

Definition of Process Management

- The kernel also tracks statistical information for the process such as the amount of time the process uses the CPU, the amount of memory the process is using, etc.
- The statistical information is used by the kernel for managing its resources and for accounting purposes.

Notes: Definition of Process Management

This page intentionally left blank.

Process Operations

- **Four basic operations define the lifetime of a process in the system:**
  - fork - *Process creation*
    - ***All processes in the system (except the boot process) are created from other processes through the fork mechanism.***
    - ***All processes are descendants of the `init` process (process 1).***
    - ***A process that forks creates a child process that is nearly a duplicate of the original parent process.***
    - ***The child has a new `proc` entry (slot), PID, and registers.***
    - ***Statistical information is reset and the child initially shares most of the virtual memory space with the parent process.***
    - ***The child process initially runs the same program as the parent process.***
  - exec - *Loading of programs in process*

Notes: Process Operations

This page intentionally left blank.

Process Operations

- *To execute a new program, a process uses the `exec` set of system calls to load the new program into memory and execute the program.*
- *Each program can successively `exec` other programs to load and execute in the process.*

- exit - *Death of process*
  - *When a program no longer needs to run or execute other programs, it can `exit`.*
  - *A program that exits causes the process to enter the zombie state.*
- wait - *Parent process notification of the death of the child process, releasing the child process' slot.*
  - *The parent process can be notified of the death of the child by waiting with a system call or catching the proper signal.*
  - *Once the parent process acknowledges the death of a child process, the child process's slot is freed.*

Notes: Process Operations

This page intentionally left blank.

Process States of Process Management

- *Idle*
  - *Processes that are being created are in the idle state.*
  - *This state is temporary until the fork mechanism is able to allocate all of the necessary resources for the creation of a new process.*
  - *Once the new child process is created, it is placed in the active state.*
- *Active*
  - *This is the normal process state.*
  - *Threads in the process can be running or ready-to-run.*
- *Stopped*
  - *Process has been stopped by SIGSTOP signal.*
  - *Process can be restarted by SIGCONT signal.*
  - *All threads are in the stopped state.*

Notes: Process States of Process Management

This page intentionally left blank.

Process States of Process Management:

- *Swapped*
  - *Process cannot run until scheduler makes it active again.*
  - *All thread are in the swapped state.*
- *Zombie*
  - *Processes that have terminated with an `exit` system call are in the zombie state.*
  - *Such processes have most of their resources freed.*
  - *However, a small part of the process remains, such as the exit value that the parent process uses to determine why the child died.*
  - *If the parent issues a `wait` system call, the exit status is returned to the parent, the remaining resources of the child process are freed, and the process ceases to exist. The slot can then be used by another newly created process.*
  - *If the parent no longer exists when a child process exits, the `init` process frees the remaining resources held by the child.*

Notes: Process States of Process Management

This page intentionally left blank.

Kernel Processes - Kprocs

- Are created by the kernel
- Have a private u-area/kernel stack
- Share "text" with the rest of the kernel
- Are not affected by signals
- Cannot use shared library object code or other user protection domain code
- Run in the Kernel Protection Domain

Notes: Kernel Processes - Kprocs

Some processes in the system are kernel processes. Kernel processes are created by the kernel itself to execute independent of threads action. Even though a kernel process shows up in the process table, through "Berkeley" ps, it is part of the kernel. The scheduler is an example of a kernel process. Kernel processes are scheduled like user processes but tend to have higher priorities.

Kernel processes can have multiple threads as can user processes.

Processes and Threads

Notes: Processes and Threads

This page intentionally left blank.

Thread/Process Relationship

- The diagram following shows how the process shares most data among the threads except that each thread has its own copy of the registers, some kernel thread specific data and a private stack. Therefore, data can be passed between threads via global variables.
- A conventional uni-threaded UNIX process can only harm itself, if incorrectly coded.
- Except for such areas as explicitly shared memory segments it cannot directly affect other processes.
- All threads in a process share the same address space, so in an incorrectly coded program, one thread can damage the stack and data areas associated with other threads in that process.

Notes: Thread/Process Relationship

This page intentionally left blank.

Thread/Process Relationship

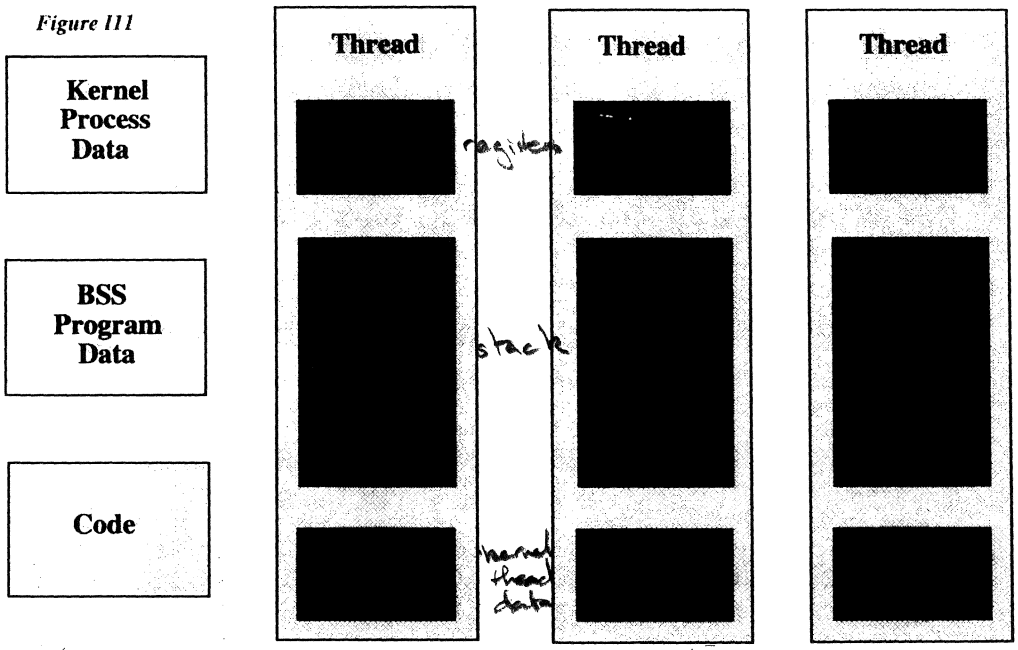
- There is some kernel data that is shared between the threads, but the kernel also maintains thread specific data.
- per-process data needed even when the process is swapped out is in the proc structure.
- per-process data needed only when the process is swapped in is in the user structure.
- per-thread data needed even when the process is swapped out is in the thread structure.
- per-thread data needed only when the process is swapped in is in the uthread structure.

Notes: Thread/Process Relationship

This page intentionally left blank.

Thread/Process Relationship

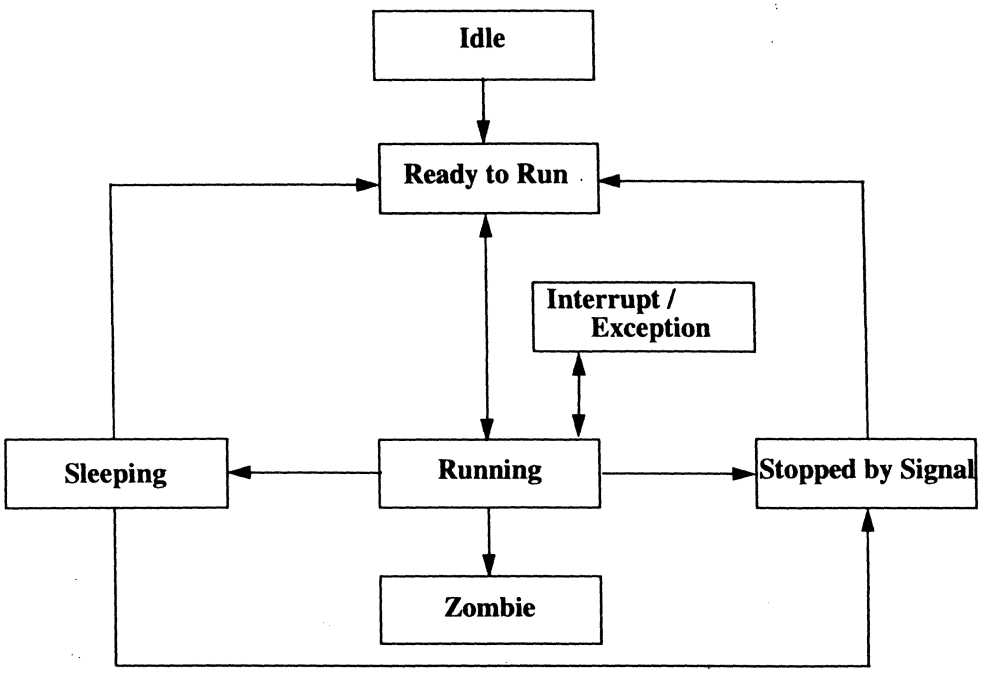
Figure 111



Notes: Thread/Process Relationship

This page intentionally left blank. left blank.

Thread States



Notes: Thread States

This page intentionally left blank.

Thread States

- In AIX, the kernel allows many threads to run at the same time, but there can be only one thread actually executing on each CPU.
- The other threads are in an inactive runable state.

Notes: Thread States

This page intentionally left blank.

Thread States

- *Running*
  - ***One thread in the system is running on each CPU. These threads are in the running state.***
  - ***While in the running state, the registers for the thread are in the machine registers.***
  - ***The CPU is executing instructions from the user virtual space or the kernel virtual space for the thread depending on the protection domain.***
  - ***A thread has a limited time slice that it is allowed to run at any one time.***
  - ***Interrupts cause the running thread to be temporarily preempted while the interrupt is processed.***
  - ***The thread can also be preempted by a higher priority thread.***

Notes: Thread States

This page intentionally left blank.

Thread States

- *Runnable*
  - *Other threads that have all the resources needed to run, but cannot run because other threads are using the CPUs, are said to be runnable.*
  - *These threads are on run queue(s) organized so that higher priority threads will run in the CPU before lower priority threads.*
  - *A thread scheduler determines thread priority.*

Notes: Thread State

This page intentionally left blank.

Thread States

- *Sleeping*
  - *Threads can be sleeping on an event.*
  - *Threads voluntarily sleep when they are waiting for a resource to become available or for data to be delivered.*
  - *Threads that are sleeping are placed on a sleep queue.*
  - *They cannot be awakened until the event on which the thread is waiting occurs, and the kernel notifies the threads.*
  - *However, some signals can awaken a thread.*
  - *When this occurs, the kernel notifies the thread by placing the thread in the runnable state on the run queue.*

Notes: Thread States

This page intentionally left blank.

Thread States

- *Idle*
  - ***Threads that are being created are in the idle state.***
  - ***This state is temporary until the fork or thread creation mechanism is able to allocate all of the necessary resources for the creation of a new process or thread.***
  - ***Once the new thread is created, it is placed in the runnable state.***
- *Stopped*
  - ***When a process is stopped by SIGSTOP, the threads are all placed in the stopped state.***
  - ***When the process is restarted with SIGCONT, the threads will be returned to their original state.***
- *Zombie*
  - ***Threads that have terminated are in the zombie state.***

Notes: Thread States

This page intentionally left blank.

Thread Scheduling

- A kernel daemon (swapper) increases thread priorities.
- Clock tick interrupt decreases the currently running thread's thread priority.
- Dispatcher chooses the highest priority thread to execute.

Notes: Thread Scheduling

This page intentionally left blank.

The Dispatcher

- **Runs under the following circumstances**
  - ***A clock tick occurs.***
  - ***A thread voluntarily gives up the CPU.***
  - ***A thread is returning to user mode from kernel mode.***
  - ***Another thread has been made runnable.***
- **A context switch consists of:**
  - ***Saving the machine state of the departing thread.***
  - ***Recalling machine state of the selected thread.***
  - ***Mapping the process private data and other virtual space of the selected thread.***
  - ***Switching the CPU to execute with the selected thread's registers.***

Notes: The Dispatcher

This page intentionally left blank.

Context Switch

- **Switches context to make a different thread execute.**
  - ***As a thread executes in the CPU, its priority becomes less favored.***
  - ***The scheduler re-calculates the priority of the executing thread and measures the new priority against the priorities of the threads that are runnable.***
  - ***In AIX, the run queues are divided into 128 separate priority queues with priority 0 being the most-favored priority and priority 127 the least-favored.***
  - ***Threads at the same priority level are on the same run queue for quick determination of the next runnable process.***
  - ***All of the threads on a more-favored priority queue run before threads on a less-favored priority queue.***
  - ***Queue 127 contains the wait threads. There is one wait thread per CPU and these run only when there are no other runnable threads.***

Notes: Context Switch

This page intentionally left blank.

Thread Preemption

- *In AIX, the kernel allows thread preemption.*
- *Preemption allows the kernel to respond to real-time processes much faster.*
- *On most UNIX systems, when a thread is in kernel mode, no other thread can execute until the thread in kernel mode returns to user mode or voluntarily gives up the CPU.*
- *In AIX, other higher priority threads may preempt threads running in kernel mode.*
- *This feature supports real-time processing where a real-time process must respond to an action immediately.*
- *Some sections of code have been determined to be critical sections where preemption is not possible because preemption may cause inconsistent kernel data structures. These sections are protected either by preventing preemption (by disabling interrupts) or by holding a lock.*

Notes: Thread Preemption

This page intentionally left blank.

Thread Preemption

- *The kernel can use locks to serialize access to global kernel data that could be corrupted by preemption.*
- *The thread holding the lock for a piece of data is guaranteed to run at a higher priority than the set of threads waiting for the lock. This is called priority promotion.*
- *However, other threads running at higher priority and not asking for the lock on the same piece of data can preempt the locking thread.*

Notes: Thread Preemption

This page intentionally left blank.

The MP Scheduler/Dispatcher

- ***Hard Cache Affinity - Ability to bind a thread or process to a processor.***
- ***Soft Cache Affinity - Attempt to run a thread or process on the same processor.***
- ***Support funneling threads.***

Notes: The MP Dispatcher/Scheduler

Scheduler:

AIX Scheduling uses a time-sharing, priority based scheduling algorithm. This algorithm favors threads that do not consume large amounts of the processor, as the amount of processor time used by the thread is included in the priority recalculation equation. Fixed priority threads are allowed. The priority of these threads do not changeregardless of the amount of processor time used.

There is one global priority-based, multi-level run queue (runq). All threads that are runable are linked into one of these runq entries. There are currently 128 priorities (0-127). The scheduler periodically scans the list of all active threads and recalculates thread priorities based on the amount of processor time used.

Threads Scheduling

- Threads are the dispatchable unit for the AIX scheduler.
- Each thread has its own priority (0-127) and scheduling algorithm.
- Three Scheduling algorithms:
  - *SCHED_RR*
    - *This is a Round Robin scheduling mechanism in which the thread is time-sliced at fixed priority.*
    - *This scheme is similar to creating a fixed priority, real time process.*
    - *The thread must have root authority to be able to use this scheduling mechanism.*

Notes: Threads Scheduling

This page intentionally left blank.

Threads Scheduling

- *SCHED_FIFO*
  - ***Non-preemptive scheduling scheme.***
  - ***Thread runs at fixed priority and is not time-sliced.***
  - ***It will be allowed to run on a processor until it voluntarily relinquishes by blocking or yielding.***
  - ***A thread using SCHED_FIFO must also have root authority to use it.***
  - ***It is possible to create a thread with SCHED_FIFO that has a high enough priority that it could monopolize the processor.***
- *SCHED_OTHER*
  - ***Default AIX scheduling.***
  - ***Priority degrades with CPU usage.***

Notes: Threads Scheduling

This page intentionally left blank.

Processes and Threads

- *Process holds address space*
- *Thread holds execution context*
- *Multiple threads can run within one process*
  - *On SMP systems, can actually run truly concurrently*

Notes: Processes and Threads

This page intentionally left blank.

Threads

- **Threads allow multiple execution units to share the same address space**
  - ***Reduces the need for IPC***
  - ***Adds complexity and risk, e.g. synchronization and locking***
- **The thread is the fundamental unit of execution**
  - ***Every process has at least one thread***
  - ***The scheduler and dispatcher only work with threads***
- **Thread IDs (TIDs) as well as Process IDs (PIDs)**
  - ***TIDs are always odd***
  - ***PIDs are always even (except for PID 1 - the init process)***

Notes: Threads

This page intentionally left blank.

Threads

- An independent flow of control within a process.
- In a multithreaded process, each thread can execute a different code concurrently.
- Managing threads needs fewer system resources than managing processes.
- Inter-thread communication is more efficient than inter-process communication.

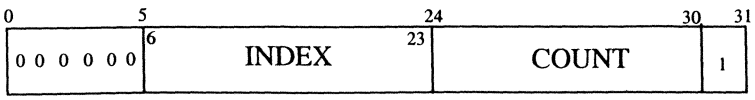
Notes: Threads

AIX 4.1 Threads:

- A thread is an independent flow of control that operates within the same address space as other independent flows of controls within a process. In other operating systems, threads are sometimes called "lightweight processes," or the meaning of the word "thread" is sometimes slightly different.
- Multiple threads of control allow an application to overlap operations such as reading from a terminal or writing to a disk file. This also allows an application to service requests from multiple users at the same time.
- Multiple threads of control *within a single process* are required by application developers to be able to provide these capabilities without the overhead of multiple processes.
- Multiple threads of control within a single process allow application developers to exploit the throughput of multiprocessor (MP) hardware.

TIDs:

- Threads IDs have the following format:



- INDEX identifies the entry in the thread table corresponding to the designated TID (thread[INDEX])
- COUNT is a generation count that is intended to avoid the rapid reallocation of TIDs. When a new TID is to be allocated, its value is predicated on the first available thread table entry. Slots are recycled.

Threads Concepts

- **Thread Safe Application:**
  - ***When multiple threads in a process can run that application sucessfully without data corruption.***
- **Kernel Thread:**
  - ***A thread of control managed by the kernel.***
- **User Thread:**
  - ***A thread of control managed by the application.***
- **In a multi-threaded system:**
  - ***The process is the swappable entity.***
  - ***The thread is the schedulable entity.***

Notes: Threads Concepts

Threads Concepts:

- A library is **thread safe** when multiple threads can be running a routine in that library without data corruption.
- A **user thread** is attached to a **kernel thread** to gain access to system services.
- The AIX Version 4.1 threads support is based on the **OSF/1 libpthreads** implementation. It supports what is referred to as the **1:1 model**. This means that for every thread visible to an application, there is a corresponding kernel thread. Architecturally, it is possible to have a M:N libptreads model, where "M" user threads are multiplexed on "N" kernel threads. This is supported by AIX 4.3.1.
- User threads are mapped to kernel threads by the threads library. The way this mapping is done is called the thread model. There are three possible thread models, corresponding to three different ways to map user threads to kernel threads.

  M:1 model

  1:1 model

  M:N model.

- The mapping of user threads to kernel threads is done using virtual processors. A virtual processor (VP) is a library entity that is usually implicit. For a user thread, the virtual processor behaves as a CPU for a kernel thread. In the library, the virtual processor is a kernel thread or a structure bound to a kernel thread.
- The libpthreads implementation is provided for application developers to develop portable multi-threaded applications The libpthreads library has been written as per the **POSIX 1003.4a Draft 10 specification** in AIX 4.3. Previous versions of aix support the **POSIX 1003.4a Draft 7 specification. libpthreads** is a **linkable user library** that provides user space threads services to an application. libpthreads_compat.a provides the **POSIX 1003.4a Draft 7 specification** pthreads model on AIX 4.3.

Process and Thread Management

- **Four main data structures for process management**
  - ***proc - maintains process state which is always required by kernel***
    - ***The proc structure is in the process table in the kernel extension segment.***
  - ***user - maintains process state which is only required when the process is running***
    - ***The user structure is in the ublock in the process private segment.***
  - ***thread - maintains thread state which is always required by kernel***
    - ***The thread structure is in the thread table in the kernel extension segment.***
  - ***uthread - maintains thread state which is only required when the thread is running***
    - ***The uthread structure is in the ublock in the process private segment.***

Notes: Process and Thread Management

Process and Thread Management:

The proc and thread structures are maintained in the kernel extension segment as a part of the process and thread tables of the kernel. Every in-use entry in these tables is pinned, so the information there is always available to the kernel. The user and uthread structures are maintained in the process private segment of the corresponding process. These structures are only pinned when the process is not swapped out. When the process is swapped out, they are unpinned.

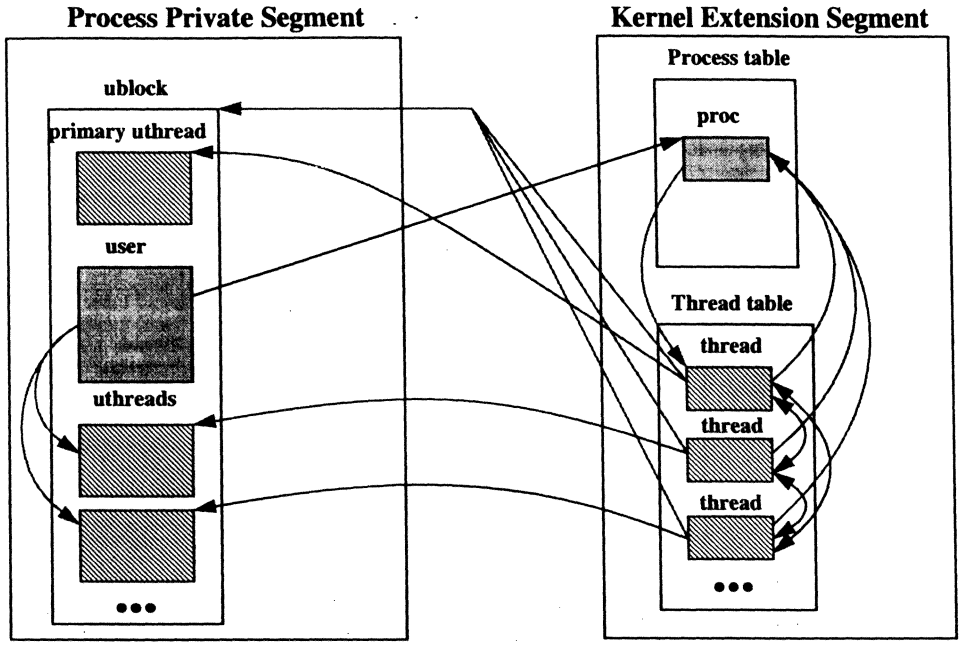
Process and Thread Management

- **Many items related to scheduling and execution state are kept in the thread-related structures**
- **All items relating to the address space go in the process-related structures**
  - ***There are also many miscellaneous items that still map to processes instead of threads - these are kept in the process-related structures, too.***
    - ***Signal handling***
    - ***nice value***
    - ***user and group IDs***
    - ***process statistics***

Notes: Process and Thread Management

This page intentionally left blank.

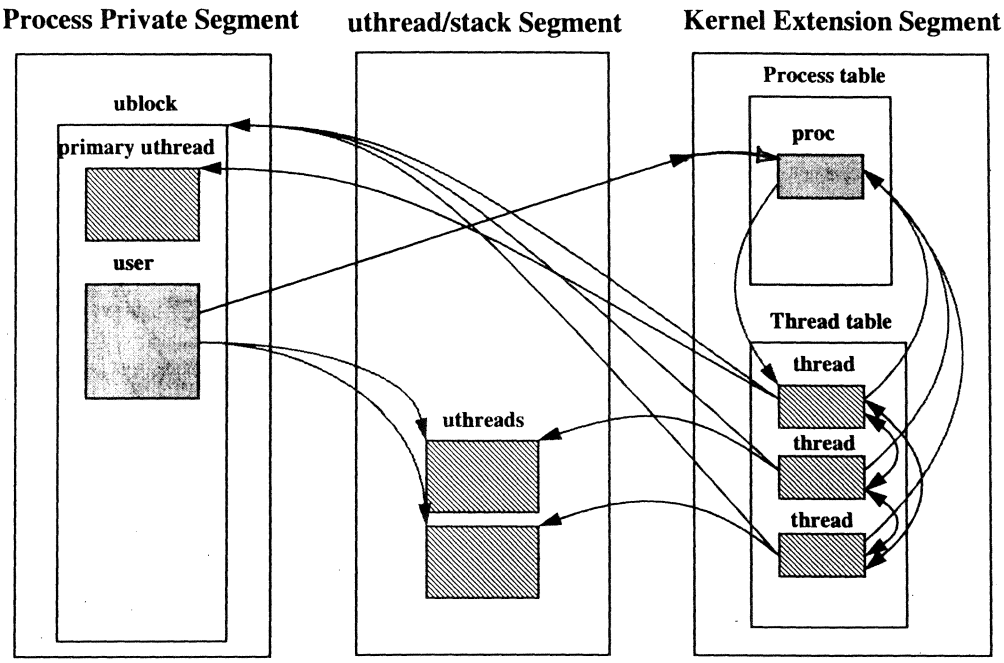
AIX Pre-4.3.1 Process/Thread Structures



Notes: Process and Thread Structures

The diagram shows that the thread structures contain pointers to all the other structures required to run that particular thread. This is a reflection of the fact that the thread is the schedulable entity. The thread structures are doubly and circularly linked to all other threads for a particular process. Note also that the ublock structure contains the user structure plus all the uthread structures for the process. The first uthread structure is kept separate within the ublock so that the fields it contains can be addressed directly.

AIX Post-4.3.1 Process/Thread Structures

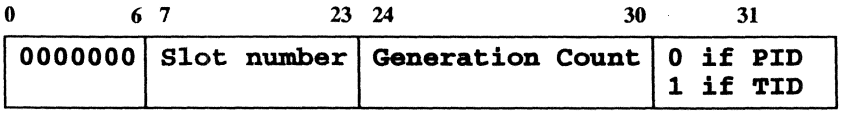


Notes: Process and Thread Structures

The diagram shows that the thread structures contain pointers to all the other structures required to run that particular thread. This is a reflection of the fact that the thread is the schedulable entity. The thread structures are doubly and circularly linked to all other threads for a particular process. Note also that the ublock structure contains the user structure plus uthread structures for the initial thread. The uthread structures for all other threads are in the uthread (and kernel thread stacks) segment. The first uthread structure is kept separate within the so that the fields it contains can be addressed directly and so that **fork** and **exec** can operate with only the process private segment to deal with.

Process and Thread Tables

- **Both kept in the kernel extension segment**
  - *Proc table starts at 0xe3000000*
  - *Thread table starts at 0xe6000000*
- **Both are maintained as arrays**
  - *Entries are called "slots"*
- **Slot number can be derived from PID or TID**



Generation count for each slot is incremented every time a PID or TID is created in that slot

Notes: Process and Thread Tables

This page intentionally left blank.

proc Structure Fields

- Defined in /usr/include/sys/proc.h
- p_stat - keeps the state of the process
  - *SNONE - Slot is not being used*
  - *SIDL - Process is being created*
  - *SACTIVE - Process has at least one active thread*
    - *May be running or sleeping*
    - *Can't be stopped or "swapped out"*
  - *SSWAP - Process is "swapped out" (i.e. suspended)*
    - *Occurs when too much contention for system resources*
    - *Not a candidate for scheduling until p_sched_count wait time expires*
  - *SSTOP - Process is stopped*
  - *SZOMB - Zombie process*

Notes: proc Structure Fields

This page intentionally left blank.

proc Structure Fields

- p_flag - large collection of process flags
  - *SNOSWAP - process can't be swapped out*
  - *SLOAD - process ublock is pinned*
    - *Shown as "swapped_in" by crash*
  - *STERM - process is terminating*
    - *Waiting for threads*
  - *Many others...*
- p_kstackseg - segment ID for the Kernel Thread Stacks segment
  NO LONGER AVAIL (AIX5 4.3.0 ONLY)
- p_adspace - segment ID for the Process Private segment

Notes: proc Structure Fields

This page intentionally left blank.

proc Structure Fields

- **Signal-related flags**
  - *3 doublewords of bit flags, one bit for each of the 64 possible signals*
  - *p_sig - Indicates which signals are pending*
  - *p_sigignore - Indicates which signals are being ignored*
  - *p_sigcatch - Indicates which signals are being caught by the process*
    - *i.e. there is a handler to call*

Notes: proc Structure Fields

This page intentionally left blank.

thread Structure Fields

- Defined in /usr/include/sys/thread.h
- t_state - keeps the state of the thread
  - *TSNONE, TSIDL, etc.*
    - *Values similar in meaning to process states*
- t_flags - large set of thread flags
  - *SWAKEONSIG - thread is sleeping while interruptible*
  - *TSIGONSTACK - thread currently running on signal stack*
  - *TSIGAVAIL - signal is available for delivery to thread*
  - *etc., etc...*

Notes: thread Structure Fields

This page intentionally left blank.

thread Structure Fields

- **t_wtype - what thread is waiting for**
  - *TNOWAIT - not waiting, currently running*
  - *TWEVENT - waiting for an event*
  - *TWLOCK - waiting for a serialization lock*
    - *Thread has slept waiting for simple version 3 lock, or write access to complex lock*
  - *TWLOCKREAD - waiting for read access to a serialization lock*
  - *TWCPU - waiting to be dispatched (ready to run)*
  - *TWPGIN - waiting for a page in*
  - *TWMEM - waiting for memory (set whenever proc is swapped out)*

Notes: thread Structure Fields

This page intentionally left blank.

thread Structure Fields

- **Signal fields:**
  - *t_sig - set of pending signals (bit flags)*
    - *Some signals can be directed to a thread*
  - *t_sigmask - set of signals blocked by thread (bit flags)*
  - *t_cursig - number of signal currently being processed*
- **Event flags:**
  - *t_pevent - set of pending events*
  - *t_wevent - set of events thread is waiting for*

Notes: thread Structure Fields

This page intentionally left blank.

M:1 Threads Model

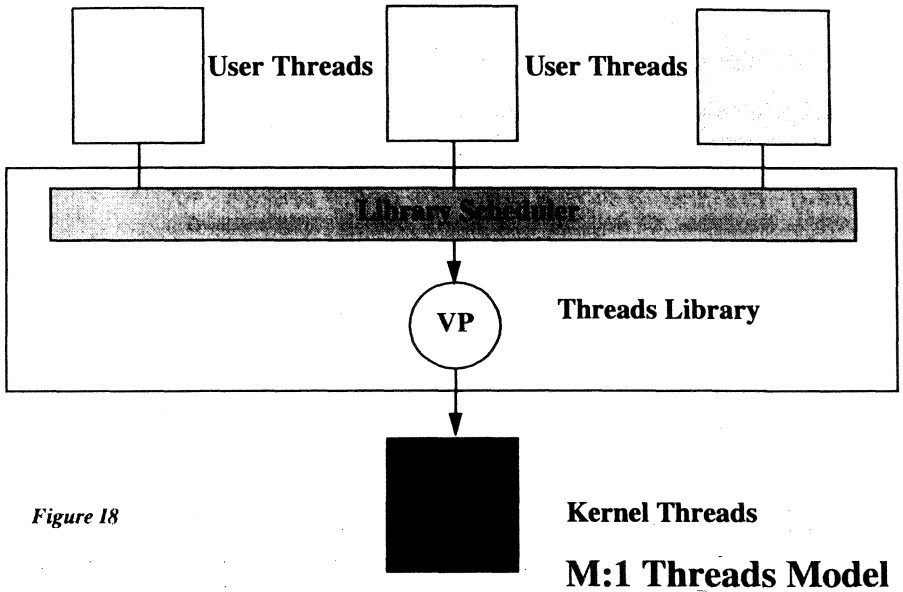


Figure 18

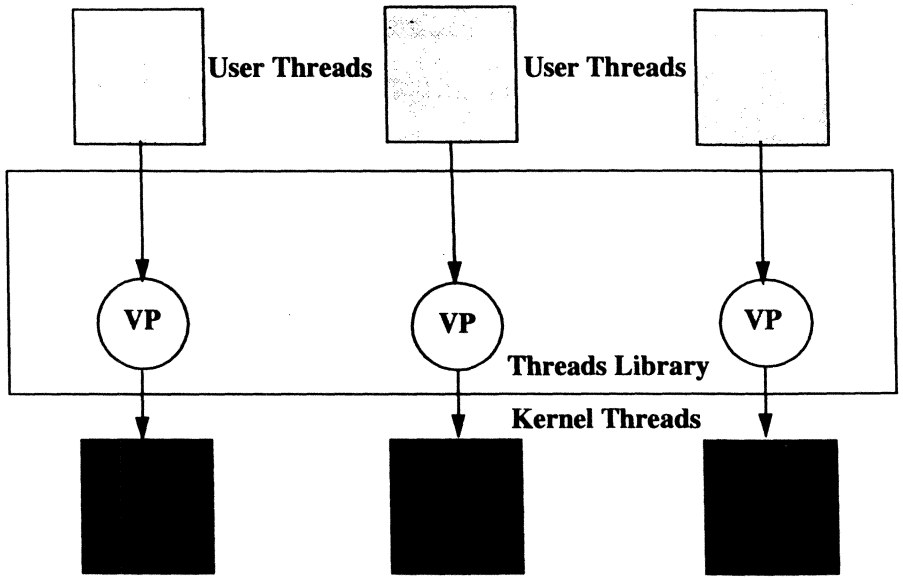
M:1 Threads Model

Notes: M:1 Threads Model

- In the M:1 model all user threads are mapped to one kernel thread; all user threads run on one VP. The mapping is handled by a library scheduler. All user threads programming facilities are completely handled by the library. This model can be used on any systems, especially on traditional single-threaded systems. The following figure I8 illustrates this model.

1:1 Threads Model

Figure I9



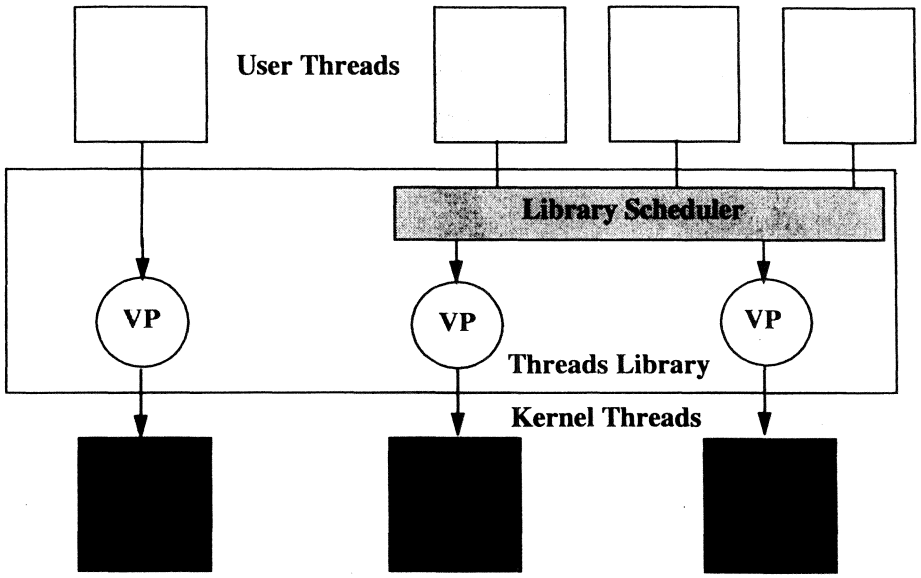
1:1 Threads Model

Notes: 1:1 Threads Model

- In the 1:1 model, each user thread is mapped to one kernel thread; each user thread runs on one VP. Most of the user threads programming facilities are directly handled by the kernel threads. The following figure I9 illustrates this model.

M:N Threads Model

Figure 110



M:N Threads Model

Notes: M:N Threads Model

- In the M:N model, all user threads are mapped to a pool of kernel threads; all userthreads run on a pool of virtual processors. A user thread may be bound to a specific VP, as in the 1:1 model. All unbound user threads share the remaining VPs. This is the most efficient and most complex thread model; the user threads programming facilities are shared between the threads library and the kernel threads. The following figure 110 illustrates this model.

Signals

- A signal is represented as a bit set in an array with enough bits set aside for each signal number.
- The bits are turned on by kernel code as the process is executing in kernel mode or by the processing of interrupts that are determined to be destined to the process.
- Signals can also be sent from one process to another process through the use of system calls.
- Delivered to the process when:
  - *Process returns to the User Protection Domain.*
  - *Transition from ready-to-run state to running state.*

Notes: Signals

This page intentionally left blank.

Signals

- To deliver a signal, the kernel checks whether the process is ~~being~~ ignored~~.~~ing it.
- If the signal is not being ignored, the kernel marks the receiving process to cause it to perform the appropriate action.
- The appropriate action may be to invoke the signal handler for that particular signal or kill the process.
- If the signal is blocked by every thread in the process, it is left pending until a thread is no longer blocking the signal.
- Signals can be delivered to a processes group.

Notes: Signals

This page intentionally left blank.

Signals

- Signals can be sent to process or thread
- Thread receives signal if
  - *Signal is synchronous and attributable to particular thread*
    - *e.g. SIGSEGV*
  - *Signal sent by thread in same process*
    - *via thread_kill system call*
- Otherwise, signal goes to process

Notes: Signals

This page intentionally left blank.

Signals to a Process

- **If signal is not being caught, signal action applies to entire process**
  - ***Every thread is terminated, stopped, or continued, depending on action***
- **If signal is being caught:**
  - ***Pick one thread that is not blocking signal to receive it***
  - ***If all are blocking, signal pending on process***

Notes: Signals to a Process

This page intentionally left blank.

Kernel Extensions

Notes: Kernel Extensions

This page intentionally left blank.

Outline

- **In-depth Kernel Extensions**
  - ***Kernel Extensions***
  - ***Device Drivers***
  - ***Kernel Processes***
  - ***Serialization and Synchronization***
  - ***Accessing Kernel Memory***

Notes: Outline

This page intentionally left blank.

Kernel Extensions

Notes: Kernel Extensions

This page intentionally left blank.

Outline

- **Kernel Extensions**
  - ***Kernel Extension Description***
  - ***Preemptable Kernel***
  - ***Pageable Kernel***
  - ***Dynamic Configuration***
  - ***Relationship to the Kernel***
  - ***Link Time Considerations***
  - ***Sysconfig System Call***
  - ***Configuration***
  - ***Programming Example***
  - ***Name Resolution***
  - ***Kernel Mode Libraries***

Notes: Outline

This page intentionally left blank.

Outline

- *Loadlist Entries*
- *Process Environment*
- *Interrupt Environment*
- *Interrupts*
- *Interrupt Priorities*
- *Exceptions*

Notes: Outline

This page intentionally left blank.

Kernel Extension Description

- **A dynamically loadable code module that adds funtionality to the kernel**
  - ***Device drivers***
    - ***LFT***
    - ***Hard disks***
    - ***CD-Rom drives***
  - ***Virtual File Systems***
    - ***AFS***
    - ***NFS***
  - ***Routines that add functionality to existing interfaces***
    - ***network routers***
    - ***security services***

Notes: Kernel Extension Description

The AIX Version 4 kernel is dynamically extendable and can be expanded by adding routines that belong to any of the following four functional classes:

- Device drivers

- Virtual file systems

- Kernel extension and device driver management routines

These extensions can be added at system boot or while the system is in operation. These modules are extensions to the kernel in the sense that they run within the protection domain of the kernel. User-level code can only access kernel extensions through the system call interface.

Kernel extensions add extensibility, configurability, and ease of system administration to AIX.

Pageable Kernel

- Allows a kernel with more functionality
- Reduces physical memory requirements
- May page fault on kernel data or code

Notes: Pageable Kernel

A major feature of AIX is the fact that its kernel is pageable. Other operating systems (such as Sun 4.03 and BSD 4.3) require that enough physical memory be available for the entire kernel to be loaded into the system at boot time.

This feature acquires more significance when considering AIX functionality may be dynamically extended using kernel extensions. Kernel extensions may be pageable, thus reducing the need for additional physical memory. But a kernel module can also pin any or all of its code or data through services provided by the AIX kernel.

Adding pageability at the kernel level is not free, however. Paging is a trade-off of flexibility with performance if code is not designed correctly.

Also, a pageable kernel adds complexity to design. Pinned code that must not page fault must not call kernel services that can page fault. If a pinned kernel service (in a kernel extension) calls an AIX kernel service that can page fault the system may crash or cause unwanted behavior.

Preemptable Kernel

- Better support for real-time systems
- May be interrupted by a higher priority thread
- Additional serialization required inside the kernel

Notes: Preemptable Kernel

In addition to pageability, the kernel is Preemptable.

Most UNIX systems incorporate a non-preemptable kernel. A thread or process entering kernel mode through a system call or trap instruction maintains control of the CPU regardless of process priority. The process maintains control until it either leaves the kernel or voluntarily sleeps, as when waiting for an I/O request to complete. Device drivers and other interrupts can preempt processes in kernel mode, but on return from the interrupt, the preempted process retains control of the CPU. In contrast, processes in user mode are always preemtable. On return from an interrupt, the kernel decides which process should run next, based on priority.

Preemptability requires additional serialization within the kernel to protect critical data structures.

Dynamic Configuration

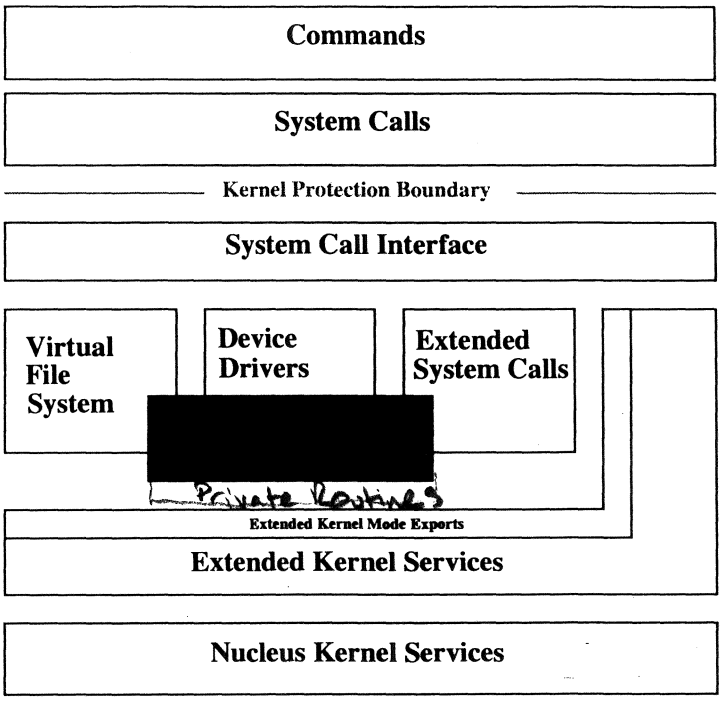
- **Advantages**
  - ***Makes system administration faster***
  - ***Extensions can be added/removed at runtime***
  - ***Reduces system impact of installation and maintenance***
    - ***requiring fewer IPLs***
    - ***system is available for longer periods of time***
  - ***Makes kernel extension development easier***
    - ***able to test new version without an IPL***
- **Disadvantages**
  - ***Allows third party code to be imported into kernel space***

Notes: Dynamic Configuration

Allowing kernel extensions to be loaded and unloaded allows a system administrator to customize a system for particular environments and applications. Rather than bundling all possible options into the kernel at compile time (and creating a large kernel), kernel extensions allow maximum flexibility. The option of loading and unloading kernel extensions at runtime increases system availability and ease of use. In addition, development time is reduced since a new kernel does not have to be compiled and installed for each development cycle.

Importing new code into the kernel allows the possibility of an unlimited number of runtime errors to be introduced into the system. Such issues as execution environment, path length, pageablity, and serialization must be taken into account when writing extensions to the kernel. This is the 'dark side' of kernel extensibility for which there is little protection.

Relationship to the Kernel



Notes: Relationship to the Kernel

Note that from the user level, access to the kernel is accomplished via the system calls exported by the kernel or by kernel extensions.

AIX allows users to extend the set of available system calls through kernel extensions. System calls can be dynamically added at runtime.

Link Time Considerations

- **Each kernel extension must have an entry point**
  - ***By convention, the*** config ***routine is the entry point for device drivers and kernel extensions***
- **Entry point, imports and exports, etc., are specified at link time**
  - ***Example:***

```
/bin/ld -H 512 -T 512 -bI/lib/kernex.exp -bE./gunk.exp \
        -lcsys -lsys -egunk_config gunk.o -o gunk
```


Notes: Link Time Considerations

Each kernel extension must have an entry point. For device drivers, this entry point is normally the configuration ("config") routine that will initialize the driver and load it into the device switch table.

As far as **ld** is concerned, kernel extensions are normal XCOFF files and can be examined using normal tools (such as **nm** and **ar**).

Sysconfig System Call

- **The** sysconfig() **system call is used to:**
  - ***Load and unload kernel extensions***
  - ***Configure and unconfigure methods for device drivers***
  - ***Invoke entry points in kernel objects***
  - ***Query status of kernel extensions***
  - ***Check status of a device driver in the device switch table***

Notes: Sysconfig System Call

Loading System Calls and Kernel Services

Kernel extensions that provide new system calls or kernel services normally place only a single copy of the routine and its static data in the kernel. When this is the case, use **SYS_SINGLELOAD sysconfig** operation to load the kernel extension. Because it only loads a new copy if one does not already exist in the kernel, this operation ensures that only a single copy is loaded. For this type of kernel extension, an updated version of the object file is loaded into the kernel only when the current copy has no users and has been unloaded.

If a kernel extension can support multiple versions of itself (particularly its data), the **SYS_KLOAD sysconfig** operation can be used. This operation loads a new copy of the object file even when one or more copies are already loaded. When this operation is used, currently loaded routines bound to the old copy of the object file continue to use the old copy. Any new routines (loaded after the new copy was loaded) are bound to the most recently loaded copy of the kernel extension.

Unloading System Calls and Kernel Services

Kernel extensions that provide new system calls or kernel services can also be unloaded. For each object file loaded, the loader maintains a usage count and a load count. The usage count indicates how many other object files have referenced some exported symbol provided by the kernel extension. The load count indicates how many explicit load requests have been made for each object file.

When an explicit unload of a kernel extension is requested, the load count is decremented. If the load count and the usage count are both equal to 0, the object file is unloaded. However if either the load count or usage count is not equal to 0, the object file is not unloaded. When programs end, the usage counts for kernel extensions that the programs referenced are adjusted. However, no unload of these kernel extensions is performed when the program ends, even if the load and usage counts become 0.

As a result, even though its load count has been decremented to 0 (due to unload requests) and its usage count has reached 0 (because of program terminations), a kernel extension can remain loaded. In this case, the kernel extension's exported symbols are still available for load-time binding unless another unload request for any object file is received. If an explicit unload request (for any program, shared library, or kernel extension) is received, the loader unloads all object files that have both load and usage count of 0.

Configuration

- **The** sysconfig() **system call is also used to invoke the entry point of a kernel extension**
- **Two options are available:**
  - **SYS_CFGDD** ***-used for device drivers***
  - **SYS_CFGKMOD** ***-used for other extensions***
- **Three sub-modes are defined for the above options:**
  - **CFG_INIT** - ***initializes an instance of a device driver or kernel extension***
  - **CFG_TERM** - ***terminates the device driver or kernel extension***
  - **CFG_QVPD** - ***queries device-specific vital product data (VPD)***

Notes: Configuration

The options described are passed as a parmeter to the kernel extension entry point. Use a switch statement or conditional to decode the parameter value.

Note that a separate value is given for device drivers. This is normally used to enter the device driver into the device switch table. It is worth mentioning that it is up to the kernel extension which parameter will have meaning. That is to say, the options have no value intrinsic to the operating system.

Programming Example

```
0 | #include <sys/types.h>
1 | #include <sys/sysconfig.h>
2 | #include <sys/device.h>
3 | #include <sys/errno.h>
4 |
5 | main(argc,argv)
6 | int argc;
7 | char *argv[];
8 | {
9 |    struct cfg_load cl;
10|    struct cfg_kmod ck;
11|    int rc;
12|    int act = -1;
13|
14|    if (argc == 3)
15|    {
16|         if ((strcmp(argv[1],"load")) == 0)
17|              act = SYS_KLOAD;
18|         else if ((strcmp(argv[1],"unload")) == 0)
19|              act = SYS_KULOAD;
20|    }
21|    if (act == -1)
22|    {
23|         printf("usage: %s load|unload <kmod>\n",argv[0]);
24|         exit(1);
25|    }
26|
```


Notes: Programming Example

This programming example is a complete program that will load and configure or unconfigure and unload a kernel extension.

Programming Example

```
27 |    cl.path = argv[2];
28 |    cl.libpath = NULL;
29 |
30 |    if (act == SYS_KLOAD)
31 |    {
32 |         /* Load kernel module.
33 |          */
34 |         rc = sysconfig(SYS_KLOAD,&cl,sizeof(struct cfg_load));
35 |         if (rc == CONF_SUCC)
36 |              printf("kload returns %d, kmid = %x\n",rc,cl.kmid);
37 |         else
38 |              printf("kload returns %d, errno = %d\n",rc,errno);
39 |
40 |         if ((rc == CONF_SUCC) && (cl.kmid != 0))
41 |         {
42 |             /* Invoke entry point (kmod) or config routine (dd)
43 |              * for kernel extension with CFG_INIT parameter.
44 |              */
45 |             ck.kmid = cl.kmid;
46 |             ck.cmd = CFG_INIT;
47 |             rc = sysconfig(SYS_CFGKMOD,&ck,sizeof(struct cfg_kmod));
48 |             if (rc == CONF_SUCC)
49 |                  printf("cfgkmod returns %d, kmid = %x\n",
50 |                       rc,cl.kmid);
51 |             else
52 |                  printf("cfgkmod returns %d, errno = %d\n",
53 |                       rc,errno);
```


Notes: Programming Example

This page intentionally left blank.

Programming Example

```
54 |          }
55 |     }
56 |     else /* unload */
57 |     {
58 |          /* Query to determine if kernel module is loaded.
59 |           */
60 |          rc = sysconfig(SYS_QUERYLOAD,&cl,sizeof(struct cfg_load));
61 |          if (rc == CONF_SUCC)
62 |               printf("queryload returns %d, kmid = %x\n",rc,cl.kmid);
63 |          else
64 |               printf("queryload returns %d, errno = %x\n",rc,errno);
65 |
66 |          if ((rc == 0) && (cl.kmid != 0))
67 |          {
68 |               /* Invoke entry point (kmod) or config routine (dd)
69 |                * for kernel extension with CFG_TERM parameter.
70 |                */
71 |               ck.kmid = cl.kmid;
72 |               ck.cmd = CFG_TERM;
73 |               ck.mdiptr = 0;
74 |               ck.mdilen = 0;
75 |               rc = sysconfig(SYS_CFGKMOD,&ck,sizeof(struct cfg_kmod));
76 |               if (rc == CONF_SUCC)
77 |                    printf("cfgkmod returns %d, kmid = %x\n",
78 |                         rc,cl.kmid);
79 |               else
80 |               {
```


Notes: Programming Example

This page intentionally left blank.

Programming Example

```
81 |                     printf("cfgkmod returns %d, errno = %d\n",
82 |                         rc,errno);
83 |                     exit(rc);
84 |                 }
85 |                 /* Unconfigure kernel module.
86 |                  */
87 |                 rc = sysconfig(SYS_KULOAD,&cl,sizeof(struct cfg_load));
88 |                 if (rc == CONF_SUCC)
89 |                     printf("unload returns %d, kmid = %x\n",
90 |                         rc,cl.kmid);
91 |                 else
92 |                     printf("unload returns %d, errno = %d\n",
93 |                         rc,errno);
94 |             }
95 |     }
96 |     exit(rc);
97 | }
```


Notes: Programming Example

This page intentionally left blank.

Slibclean

- **The** slibclean **command removes kernel modules from memory**
  - ***Removes uneeded object files from shared library region and the kernel***
  - ***Reduces memory fragmentation***

Notes: Slibclean

The **slibclean** command, which unloads all object files with load and use counts of 0, can be used to remove object files that are no longer used from both the shared library region and the kernel. Periodically invoking this command reduces the effects of memory fragmentation in the shared library and kernel text regions by removing object files that are no longer required.

Name Resolution

- **Import and Export Files**
  - ***An import or export file is an ASCII file consisting of symbol names, one per line***
  - ***The export file of a module becomes the import file of another module that requires routines from the former module***
  - ***Kernel services exported by the kernel are available in the export file*** **kernex.exp**
  - ***System calls exported by the kernel are available in the export file*** **syscalls.exp**

Notes: Name Resolution

The kernel provides a set of base kernel services to be used by kernel extensions. These services, which are described in the services documentation, are made available to a kernel extension by specifying the **kernex.exp** export file as an import when linking the extension. The linking operation is performed by using the **ld** command.

A kernel extension provides additional kernel services and system calls by supplying an export file when it is linked. This export file specifies the symbols to be added to the **/unix** name space, which is the global kernel name space. Symbols that name system calls to be exported must specify the **syscall** keyword next to the symbol in the export file.

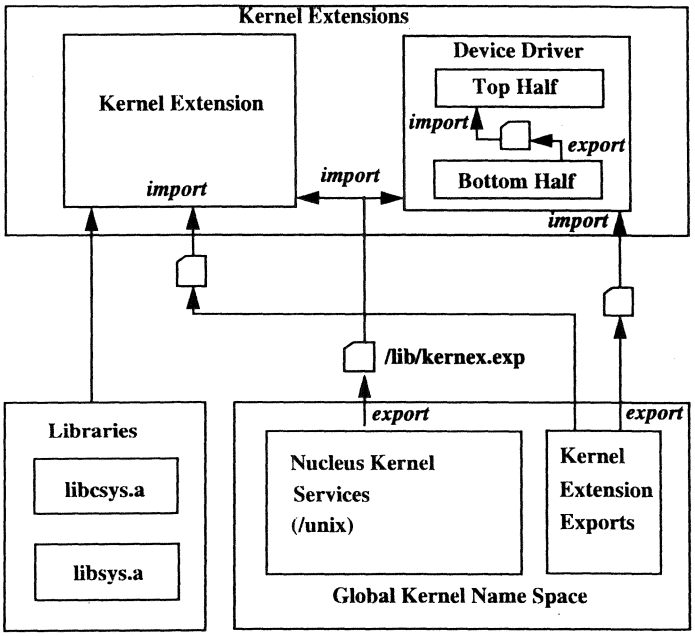
The kernel extension export file must also have **#!/unix** as its first entry. The export file can then be used by other extensions as an import file. The **#!/unix** as the first entry in an import file specifies that the imported symbols are to come from the **/unix** name space. This entry is ignored when used in an export file. The same file can be used both as the export file for the kernel extension providing the symbols and as the import file for another extension importing one or more of the symbols.

When a new kernel extension is loaded by the sysconfig subroutine, any symbols defined in the extension export file at link-edit time are added to the **/unix** kernel name space. The loader can also load additional object files into the kernel to resolve symbols referenced by the new extension. Because these exported symbols are only used to resolve references required during loading of the new extension, these additional object files will not have their own exported symbols added to the name space.

In other words, the kernel name space cannot be expanded without the explicit loading of a kernel object file specifying one or more exported symbols. The symbols added to the kernel name space are available to any subsequently loaded kernel object file as an imported symbol.

An object file explicitly loaded into the kernel exporting symbols into the kernel name space is shared by all kernel extensions. Normally, only one copy of the object file exists in the kernel.

Name Resolution



Notes: Name Resolution

Kernel extensions can also consist of several separately linked object files that are bound at load time. This is particularly useful for device drivers, where one object file contains the top (pageable) half of the driver and a second object file contains the bottom (pinned) half of the driver. Load-time binding is useful where several kernel extensions use common routines provided in a separate object file.

In both cases, the symbols exported by the private object files should not be added to the global kernel name space. If it is to have certain symbols exported to the global kernel name space and use other symbols only to resolve references to other private object files, the kernel extension should be divided into separately link-edited object files. One object file would contain the symbols to be exported to the kernel name space, while the other would contain the exported symbols that are considered private.

For object files that reference each other's symbols, each file should use the other's export file as its own import file during link-edit. The export file for the object file providing the services should specify **#!<filename>** as the first entry in the export file. When used as an import file, however, the entry tells the loader where to find the object file that resolves the imported symbols at load time.

The object file that exports symbols to the kernel name space must specify **#!/unix** as the first entry in its export file. This allows the export file to be used as an import file by other kernel extensions. The object file containing the symbols to be exported to the kernel name space must be the one explicitly loaded into the kernel with the **sysconfig** subroutine. The loader then loads other private object files, as necessary, to resolve imported symbols required for the load.

When, during the same explicit load request, the loader encounters an imported symbol that is resolved by an already loaded object file, the loader does not load a new copy. Instead, it resolves the symbol to the copy of the already loaded object file. This allows for cross-resolving symbols between two or more object files loaded as a result of the same explicit load request. The loader hashes the path and file name of the object file to determine whether the file has already been loaded during this explicit load request. Another copy of the object file can be loaded if differing path names are used for the same object file and the town ames do not hash to the same value.

Name Resolution

Export file format example (kernex.exp):

```
#!/unix
* comments start after first line!

*****************************************
* Exported global data structures
*****************************************
kernel_heap
kernel_lock
pinned_heap
tod
audit_flag

*****************************************
* Exported mbuf services
*****************************************
net_lock
m_collapse
m_adj
m_cat
```


Notes: Name Resolution

This page intentionally left blank.

Name Resolution

Export file format example (syscalls.exp)

```
#!/unix
* comments after first line!

*       syscalls.exp -- system call entry points
*
access          syscall
accessx         syscall
acct            syscall
adjtime         syscall
audit           syscall
auditbin        syscall
```


Notes: Name Resolution

This page intentionally left blank.

Kernel Mode Libraries

- **Two libraries are provided for kernel extensions:**
  - **libcsys.a** - ***provides subset of routines found in libc.a***
  - **libsys.a** - ***provides routines not available at user level***
- **Must be bound to kernel extension during linking phase**
- **Example:**

```
/bin/ld -b binder:/usr/lib/bind glink:/lib/glink.o \
-bI/lib/syscalls.exp ./gunk.exp \
 -lcsys -lsys \
-egunk_config gunk.o -o gunk
```


Notes: Kernel Mode Libraries

A library is a collection of previously linked object files or import files and is created by using the **ar** (archive) command. Each object file or import file within the archive (library) is referred to as a member. Program management allows a member (or object file) to be designated as shared when it is linked. Libraries with or without shared objects can be created and used by kernel extensions. However, due to the different programming requirements in the kernel, library services provided for user-mode applications generally should not be used by kernel extensions.

When it resolves a symbol to a library member (or object file) not designated as shared, the linker (**ld** command) binds the required object file into the output object file so that the references will resolve. However, when symbols are resolved to a library member (or object file) designated as shared, the shared object file is not included in the output object file. Instead, the linkage editor adds information to the loader section of the output object file. The loader uses this information at load time to find the location of the shared object file that resolves the symbol.

When these shared object files (normally in libraries) are referenced by user-mode programs, the loader checks the shared library region to determine if the object file is in the shared library region. If it is, the references are resolved to the object file in the shared library region. If the object file has not already been loaded, the loader will load it into the shared library region if the file permissions allow it. In this way, common or shared object files used by user-mode applications can be shared by all user-mode programs in the system.

Unlike user mode, the kernel does not provide a shared library region. Therefore, when a kernel extension that refers to a shared object file is loaded, the loader loads a new copy of the shared object file into the kernel to be used to resolve all references to the object file during the explicit kernel extension load request. However, within the same explicit load request, all references to the *same* object files are resolved to the single copy of the object loaded for the current load request.

System Calls in Kernel Extensions

- **The normal system call interface is not available to kernel extensions**
  - ***No need - already in kernel mode, so no mode switch is necessary***
- **Extensions can call system calls more directly**
  - ***When loaded, references to exported system call symbols are resolved to point directly to a real function descriptor for the kernel routine***
  - ***When called, extension just uses the normal kernel glink code***
  - ***The entire system call interface is bypassed***
- **However, most system calls cannot be called from kernel extensions running under user processes**
  - ***If there are parameters passed by reference, they must be user addresses.***

Notes: System Calls in Kernel Extensions

This page intentionally left blank.

Loadlist Entries

- **Use** le **command in crash for information about kernel extensions**
- **Example loadlist entry:**

```
LoadList entry at 0x0a150180
 Module start:0x019f5000  Module filesize:0x00005f68  *end:0x019faf68
 *data:0x019fada0  data length:0x000001c8
 Use-count:0x000b  load_count:0x000b  *file:0x00000000
 flags:0x00000262 TEXT DATX DATA DAEX
 *exp:0x00000000  *lex:0x00000000  *deferred:0x00000000
 Name: /etc/drivers/scdisk
 ndepend:0x0002  maxdepend:0x0002
 *depend[00]:0x055eb040
 *depend[01]:0x0a150380
 le_next:  0a150280
```


Notes: Loadlist Entries

The **/unix** entries correspond to:

1. Symbols exported by the kernel that can be imported/used by other kernel extensions.

2. Symbols exported as system calls that can be imported/used by application programs.

The entries named **/unix** in the kernel load list represent a new version of the kernel name space. Each time a kernel extension is loaded that exports symbols, a new instance of the **/unix** is created.

use-count: incremented each time a module which depends on the loader entry is created.

file: pointer to object file (beginning with a magic number) loaded in memory.

data: pointer to initialized data area. Note: data length includes uninitialized data.

exp: list of exported symbols associated with kernel extension.

lex: temporary symbols used only at load time.

deferred: pointer to list of symbols marked for deferred resolution.

ndepend: number of loader entries this entry depends on.

maxdepend: used internally by the loader.

depend[ ]: actual list of modules this module depends on.

Process Environment

- Not running in an interrupt handler or in code called by an interrupt handler.
- Can have interrupts disabled by explicit call to disable to some interrupt priority.
- Can be running under a user process or kernel process
- getpid() returns process ID

Notes: Process Environment

A process in kernel mode can also put *itself* into an environment similar to the interrupt environment. This action, occurring when the interrupt priority is changed to a priority more favored than INTBASE, can be accomplished by calling the **disable_lock** or **i_disable** kernel services. A kernel-mode process is sometimes required to do this to serialize access to a resource shared by a routine executing in the interrupt environment. When this is the case, the process operates under most of the same restrictions as a routine executing in the interrupt environment. One exception is that there is still a process context available, so the available address space (i.e. segment register setup) is the same as it was before the disable occurred. However, page faults are still not be allowed.

Routines executed in this environment with interrupts disabled can adversely affect system real-time performance and should therefore be limited to a specific maximum path length. Path length is defined as the processor cycles required to execute at a particular interrupt level.

Base Execution Level

- Running in process environment
- Executing at an interrupt priority of INTBASE
- May cause page faults by accessing pageable code or data
- May be preempted by another process of equal or higher priority
- May sleep

Notes: Base Execution Level

Routines running base execution level are executed at an interrupt priority of INTBASE (the least favored priority). Code running at this level can cause page faults by accessing pageable code or data. It can also be preempted by another process of equal or higher process priority.

A routine running at the base execution level can sleep or be interrupted by routines executing in the interrupt environment. A kernel process, however, can use all system calls listed in the *Kernel Extensions and Device Support* book.

Interrupt Environment

- Called on behalf of an interrupt handler
- Must not cause page faults when accessing code or data
- Must not sleep (block)
- Limited Stack Size
- No process context
  - *Only segments 0, 5, 7, 8, 9, 10, and 14 mapped*
    - *Other segments may be left behind by last running process before interrupt*
  - *getpid() returns -1*

Notes: Interrupt Environment

A routine runs in the interrupt environment when called on behalf on an interrupt handler. A kernel routine executing in this environment cannot request data that has been paged out of memory and therefore cannot cause page faults by accessing pageable code or data. In addition, the kernel routine has a stack of limited size, is not subject to preemption by another process, and cannot perform any function that would cause it to sleep. Also, note that the only segments guaranteed by the system are segment 0 (the kernel segment) and segment 14 (the kernel extension segment). Any other segment registers are either cleared, or left as they were for the last process that was running when the interrupt occurred. It is very unlikely that the running process at the time of the interrupt knows anything about the interrupt, so interrupt handlers need to be careful to avoid using other segments.

A routine in this environment is only interruptable either by interrupts that have priority more favored than the current priority or by exceptions. These routines cannot use system calls and can use only kernel services available in both the process and interrupt environments.

Routines executed in this environment can adversely affect system real-time performance and should therefore be limited to a specific maximum path length. Path length is defined as the processor cycles required to execute at a particular interrupt level.

Interrupt Processing

- **Device drivers use AIX kernel services to identify**
  - ***The interrupts they need to handle***
  - ***The priorities for those interrupts***
  - ***The specific hardware signal or level used for this interrupt***
- **AIX programs the hardware interrupt controller accordingly**
- **When a device raises an interrupt, the controller prioritizes it according to its programming**
- **When an interrupt is to be delivered, the interrupt controller notifies the processor**

Notes: Interrupt Processing

Interrupt Processing

When an external interrupt is first detected, the system immediately calls the external interrupt first-level interrupt handler (FLIH), which queries the hardware registers to determine the type of interrupt request. At this point on POWER and POWER2 machines, the interrupt is directly serviced. On PowerPC machines, however, the FLIH will enqueue the interrupt based on level and priority. This raises a flag indicating that there is pending work to be done and it will be serviced later, thus the PowerPC essentially enqueues all interrupts.

The kernel detects queued interrupts at various key times, such as when enabling to a less-favored priority from a more favored one. Once a queued interrupt is detected and the processor is executing at (or about to be enabled to) a priority that lets the pending interrupt be serviced, the current machine state is saved, and interrupt processing is started.

Then the kernel begins calling the interrupt handlers that are registered at the specified level. Because interrupt levels can be shared on certain buses, the adapter which caused the interrupt is not necessarily known at this stage. The order in which the kernel calls interrupt handlers at a certain level is the order in which they were initially registered. This ordering does not change as long as the interrupts are registered.

Once your second-level interrupt handler (SLIH) is called, it must determine whether the associated adapter caused the interrupt. If the interrupt was caused by the adapter, the interrupt handler does its necessary work, possibly schedules more off-level work, and returns INTR_SUCC. if the interrupt was not caused by the adapter, INTR_FAIL is returned, and the kernel calls the next handler in the list.

Interrupt Processing

- The processor checks the external interrupt line between execution of instructions
- If active, it branches to 0x500
- AIX first level interrupt handler takes over

Notes: Interrupt Processing

This page intentionally left blank.

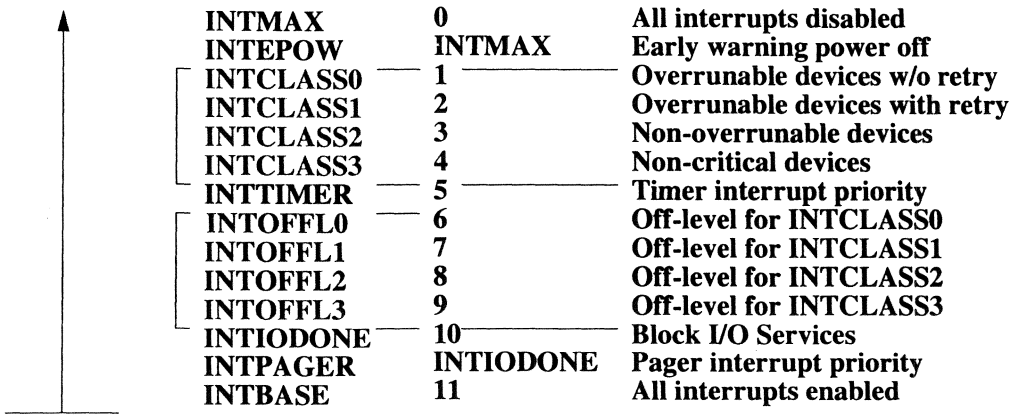
Interrupt Processing

- AIX manages many interrupts itself
  - *Enqueues interrupts based on priority and level*
- Once interrupt can be delivered, AIX finds the list of registered handlers to call
  - *May be more than one because interrupt levels may be shared*
- Each handler must check to see if the interrupt belongs to it
  - *If so, when done processing, handler should return INTR_SUCC*
  - *If not, should immediately return INTR_FAIL*

Notes: Interrupt Processing

This page intentionally left blank.

Interrupt Priorities



Notes: Interrupt Priorities

The interrupt priority defines which of a set of pending interrupts is serviced first. INTMAX is the most favored interrupt priority and INTBASE is the least favored interrupt priority. Note that a more favored interrupt priority has a lower numerical value.

The interrupt priorities for bus interrupts range from INTCLASS0 to INTCLASS3. For example, the SCSI device driver operates at INTCLASS2, which indicates that it is a non-overrunnable device. In other words, the interrupt priority that a device driver runs at is chosen depending on the type of device it controls. The more time critical a device, the move favored the priority it must operate at. Therefore, it must remain for a shorter period at that interrupt level so it will not affect the reliability of other devices (or system performance). In general, interrupts with a short interrupt latency time must have a short interrupt service time.

Interrupts that cannot be serviced within the time limits specified for bus interrupts require the use of *off-level interrupts*. These are the INTOFFLn priority levels. Typically, off-level interrupt priorities are used when the interrupt service time for an operation exceeds the time allowed at that interrupt priority. Off-level interrupts are scheduled by the more-favored interrupts; this allows other device interrupts to preempt the operation of the off-level handler at a small cost of additional system overhead.

A device's interrupt priority is selected based on two criteria: its maximum interrupt latency requirements and the device driver's interrupt execution time. The interrupt latency requirement is the maximum time within which an interrupt must be serviced. If it is not serviced in this time, some event is lost or performance is degraded seriously. The interrupt execution time is the number of machine cycles required by the device driver to service the interrupt.

Device Drivers

Notes: Device Drivers

This page intentionally left blank.

Outline

- **Device Drivers**
  - *Device Driver Description*
  - *Device Driver Classes*
  - *Adapter Drivers*
  - *Pseudo-device Drivers*
  - *Device Special Files*
  - *Device Switch Tables*
  - *Device Objects*
  - *Device Driver Roles*
  - *Top-half Considerations*
  - *Bottom-half Considerations*

Notes: Outline

This page intentionally left blank.

Device Driver Description

- Kernel extensions that manage specific devices used by the operating system
- Provides a common I/O Model (read(), write(), ioctl()).
- Shields the operating system and user code from device-specific details
- Supports devices that may be physical or virtual

Notes: Device Driver Description

Device drivers are kernel extensions that control and manage specific devices used by the operating system. The I/O subsystem, in conjunction with the device drivers, allow processes to communicate with peripheral devices such as terminals, printers, disks, tape units, and networks. Device drivers can be installed into the kernel to support a class of devices (such as disks) or a particular type of device (such as a specific disk drive model). Device drivers shield the operating system form device-specific details and provide a common I/O model for accessing the devices for which they provide support.

Device driver routines providing support for physical devices typically run in two different types of environments, thus leading to a two-part structure. One part, referred to as the *top half* of the device driver, always runs in the process environment. Routines in the top half typically provide the device head role, since they always run in the environment of the calling process.

The other part, referred to as the *bottom half* of the device driver, runs in the process or interrupt environment. Routines in the bottom half typically provide the device-handling role because they deal with actual device I/O typically driven by hardware interrupts. For block devices, the strategy routine is found in the bottom half since it can be called in the interrupt environment due to paging or other asynchronous requests.

The operating system also supports and uses the concept of virtual devices, or *pseudo-devices*, which may not have a one-to-one relationship with a physical device (or no corresponding physical device). Pseudo-devices are discussed later in this section.

Device Driver Classes

- **AIXv4 supports four classes of device drivers:**
  - ***Block***
    - ***Supports random access devices with fixed-size data blocks***
  - ***STREAMS***
    - ***Has routines that are either invoked from a stream head or a STREAMS module, instead of from the device switch (devsw) table***

Notes: Device Driver Classes

Block Device Drivers

Devices usually supported by a block device driver include: hard disk drives, diskette drives, CD-ROM readers, and tape drives. Block device drivers often provide two ways to access a block device:

raw access: The buffer supplied by the user program is to be pinned in RAM as is.

block access: The buffer supplied by the user program is to be copied to, or read from buffer in the kernel.

If the block device is accessed as *raw*, the driver can copy data from the pinned buffer to the device. In this case, the size of the buffer supplied by the user must be equal to, or some multiple of, the device's block size. The special file's name is usually prefixed by the letter **r** so that a user can tell which access type the block device has. For example, the name of a diskette drive's raw block special file is **rfd0**, and a special file name for a tape drive is **rmt0**. Sometimes the term *character mode access* is used to mean raw access.

STREAMS Device Drivers

A stream is a linked list of kernel modules, and consists of a stream head at one end of the list and a STREAMS device driver at the other. The stream head (supplied with the operating system as part of STREAMS, a device driver writer does not need to write a stream head) contains some routines that are invoked from the device switch table, so the stream head is associated with a device special file in the AIX file tree. A STREAMS driver has some routines that are either invoked by the stream head, or by a STREAMS module that had been inserted into the stream between the stream head and the STREAMS driver. The driver may (or may not) have any routines that are invoked from the device switch table.

Devices that may be supported by a STREAMS driver include: any device connected to the serial port (such as a terminal), or any device attached to a LAN or WAN (such as an Ethernet adapter). Such devices lend themselves to support from STREAMS drivers because the STREAMS facility is flexible and modular. These qualities that are well suited to implementing communication protocols.

Since the TTY subsystem in AIX Version 4.1 consists of STREAMS modules, if you want to support terminal processing from a serial adapter you must provide a STREAMS driver.

Device Driver Classes

- *CDLI*
  - *Network device driver that allows sockets and streams protocols to coexist and share the same device driver*
  - *Does not use devsw*
  - *No special file in /dev*
- *Character*
  - *Any driver that is not one of the above types*

Notes: Device Driver Classes

CDLI (Common Data Link Interface) Device Drivers

CDLI is a framework for network device drivers and data link providers. It allows sockets and streams protocols to coexist and share a single device driver. It is independent of both sockets and streams, and it only provides function that is common to both. General users do not access CDLI devices directly, but instead use sockets and Data Link Provider Interface (DLPI) data link layers.

These devices do not use the device switch table, there are no special files (entries in /dev) for them, nor are they accessed via the standard open/read/write/close mechanisms. Instead, they are maintained in a linked list of network device driver (ndd) structures, and they are accessed via a specially-defined set of kernel services:

ns_alloc
ns_free
ndd_output
ndd_ctl
...

CDLI is covered in great detail in the AIX V4.1 Networking Workshop.

Character Device Drivers

Devices that are supported by a character device driver include any device that reads or writes data a character at a time (such as printers, sound boards, or terminals). Also, any driver that has no associated hardware device (called a pseudo-driver) is treated as a character device driver. For example, **/dev/mem, /dev/kmem,** and **/dev/bus0** are *character pseudo-drivers.*

Adapter Drivers

- Deal directly with RISC System/6000 hardware adapters
- Use AIX kernel services to handle hardware interrupts
- Examples:
  - *SCSI adapter*
  - *keyboard*
  - *mouse*
  - *ethernet*
  - *token ring*

Notes: Adapter Drivers

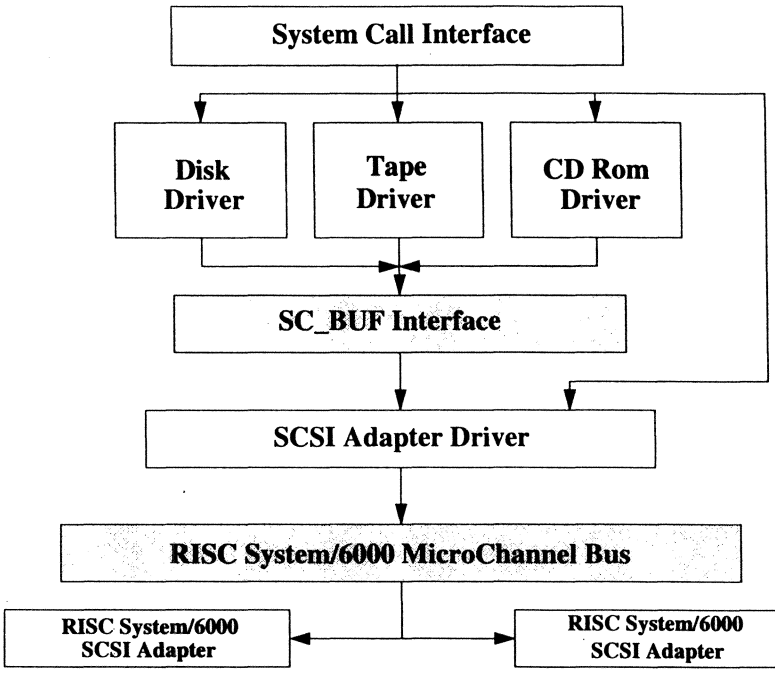
Adapter Driver Overview

An adapter driver is responsible for manipulating a Microchannel or PCI card (or other system hardware) and managing transfers to and from the card. Often the adapter driver is the only driver required for a subsystem.

Some subsystems (SCSI, LFT, ethernet, and so on) split the functionality into several pieces. In such a case, the adapter driver runs the card and manages the low level bus; the upper level driver (disk, tape, CD-ROM, and so on) manages the terminal device or virtual device using the service provided by the adapter driver. The upper level has no knowledge of any RISC System/6000 or adapter-specific requirements because they are hidden by the adapter driver. Therefore, the head can run on any compatible adapter.

For example, the SCSI disk driver works on both the Model 220 integrated SCSI adapter and the Microchannel SCSI card without modification, even though the physical adapters are very different.

Adapter Drivers



Notes: Adapter Drivers

This page intentionally left blank.

Pseudo-Device Drivers

- Manipulate virtual rather than physical devices
- Provide interfaces to aggregate devices that might otherwise require user-level code
- Examples include:
  - *LFT driver*
  - *Network PTY driver*

Notes: Pseudo-Device Drivers

Pseudo-devices are used where a set of several physical devices must be used as an integrated set. By writing device drivers for each physical device, the development process is reduced to manageable pieces. Providing a high-level virtual interface to the pieces (with a pseudo-device) enables the user or application code to run without knowledge of physical device specifics.

The LFT is the most common example of a pseudo-device driver. Almost every RISC System/6000 has a keyboard, mouse, and display. The individual hardware drivers are accessed only by the LFT, and the user application accesses the LFT pseudo device. The LFT hides the specific display characteristics in a device-independent way to ease application development.

When separate device drivers for each physical device operate to a common interface design, LFT can provide integrated device-independent support to user applications.

Pseudo-devices can also be "software devices." A good example is the pty (pseudo-terminal).

Device Special Files

- **Block and character devices are accessed through special files located in the /dev directory**
- **Special files are created with the** mknod **command**
  - ***Major and minor numbers indicate which device driver and device***
  - ***Special file types indicate class of device***

```
# ls -l /dev/hdisk0 /dev/rhdisk0
brw-------   1 root     system    18,  0 Jul 20 1994  /dev/hdisk0
crw-------   2 root     system    18,  0 Jul 20 1994  /dev/rhdisk0

# ls -l /dev/log
srw-rw-rw-   1 root     system          0 Jan  3 08:35 /dev/log
```


Notes: Device Special Files

The user interface to block and character devices goes through the file system. Every device has a name that looks like a file name and is accessed like a file. The device special file has a disk inode and occupies a node in the directory hierarchy of the file system. The device file is distinguished from other files by the file type stored in its inode, either "block," "character special," or "streams," corresponding to the device it represents.

The kernel contains many entry points into the file I/O subsystem. Common entry points are called using the **open, close, read, write, lseek,** and **ioctl** subroutines. The file I/O subsystem determines whether the request is to gain access to an ordinary file, a block special file, or a character special file. In the case of device special files, this subsystem translates the file name into major and minor numbers, which are used to select the device driver and a specific device respectively.

Device special files do not have to be created every time a device driver is loaded. It is only necessary to create them when you first add a new device to the system. Device special files are created with the **mknod** command. For example, adding a device special file for a tty might look like this:

```
mknod /dev/tty2 c 16 2
```

This specifies that the device /dev/tty2 is a character special file, 16 is the major number and 2 is the minor number. The major number indicates a device type that corresponds to the appropriate entry in the device switch table (explained shortly), and the minor number indicates an instance of the device. If a thread opens the character special file /dev/tty2 and its major number is 16, the kernel calls the open function in slot number 16 of the device switch table.

Device Switch Table

- Block and character device driver routines accessed through *devsw* table
- Each devsw element is a table of entry points (addresses)
- Entry points inserted into devsw at config time
- Each table entry is indexed by a *major number*

Notes: Device Switch Table

The Device Switch Table

The file system accesses block and character device driver routines through a table called the device switch table. This table is kept in kernel storage and contains one element for each configured device driver. Each element is itself a table of entry point addresses with one address for each entry point provided by that device driver.

Device driver entry points are inserted in the device switch table at device driver configuration time. The driver configuration routines call various kernel services to install driver entry points into one or more entries (rows) of the table. Each table entry or row is indexed by a major number.

Major numbers are assigned at device configuration time by the configuration management routines used by device configuration methods (in particular, the **genmajor** device configuration subroutine). The major number assigned to a device driver for its entry into the device switch table is the same as the major number in the device special file associated with the device.

Major and Minor Numbers for a Special File

Devices are generally identified in the kernel through major and minor numbers. Usually, a major number identifies a particular device driver. Minor numbers identify specific device instances known to the device driver. However, a device driver can be assigned multiple major numbers. Also, minor numbers can be used to identify different modes of operation for a specific device as well as different device instances.

Programs do not need to understand these major and minor numbers to access devices. A program accesses a device by opening the corresponding device special file located in the **/dev** directory. The special file disk inode contains a particular major and minor number combination specified when the special file was created. This relationship remains constant until the special file is deleted.

The major number uniquely identifies the relevant device driver and thus is used to index into the device switch table maintained by the kernel. The interpretation of the minor number is entirely dependent on the particular device driver. Most frequently, the minor number is used to select one of many actual devices supported by the device driver. The minor device number usually serves as an index into a device driver-maintained array of information about each of many

Device Switch Table

```
#define DEVCNT  256          /* # of entries in device switch tbl   */
/*  Device Switch Table entry structure. */
struct devsw {
      int    (*d_open)();   /* entry point for open routine       */
      int    (*d_close)();  /* entry point for close routine      */
      int    (*d_read)();   /* entry point for read routine       */
      int    (*d_write)();  /* entry point for write routine      */
      int    (*d_ioctl)();  /* entry point for ioctl routine      */
      int    (*d_strategy)();/* entry point for strategy routine  */
      struct tty     *d_ttys;/* pointer to tty device structure   */
      int    (*d_select)(); /* entry point for select routine     */
      int    (*d_config)(); /* entry point for config routine     */
      int    (*d_print)();  /* entry point for print routine      */
      int    (*d_dump)();   /* entry point for dump routine       */
      int    (*d_mpx)();    /* entry point for mpx routine        */
      int    (*d_revoke)(); /* entry point for revoke routine     */
      caddr_t d_dsdptr;     /* pointer to device specific data    */
      /*
       *    The following entries are control fields managed
       *    by the kernel and should not be modified by device
       *    drivers or kernel extensions.  They may be set to
       *    0 in the devsw structure for devswadd(), since they
       *    are not copied into the device switch table.
       */
      caddr_t d_selptr;     /* ptr to outstanding select cntl blks */
      ulong  d_opts;        /* internal device switch control field */
};
```


Notes: Device Switch Table

This page intentionally left blank.

Device Driver Objects

- **Device driver modules located in** /usr/lib/drivers
- **May be split in two parts:**
  - ***top half (pageable) - executes in process environment***
  - ***bottom half (pinned) - executes in interrupt environment or code that serializes with the interrupt environment***
- **Naming convention**
  - ***xyz - top half or complete driver (including the bottom half)***
  - ***xyzpin or xyzbot - bottom half***

Notes: Device Driver Objects

Sometimes a driver routine (such as an interrupt handler) and any associated data is required to be kept in memory. This requirement might exist to avoid handling a page fault so the routine can execute within a fixed period of time or to avoid receiving a page fault while interrupts are disabled.

Typically, a device driver writer compiles or links all routine and data definitions that must be pinned into one loadable file (the bottom half of the device driver), because the **pincode** kernel service is used to pin the module. **pincode** marks each page of the loaded module as being required to be kept in RAM. A pointer to a routine within the object file is the input value for the **pincode** kernel service.

Routines and data that can be subject to page replacement are typically collected into another loadable object (the top half of the device driver).

Routines that wish to allocate buffers from the kernel heap (by calling the **xmalloc** kernel service) must take care which heap the allocation is from. Routines in the bottom half allocate data from the pinned heap; and routines in the top half allocate data from either the kernel heap or the pinned heap.

Device Driver Roles

- **Drivers play two roles in AIX Version 4:**
  - ***Device head - handles requests generated by base level code.***
  - ***Device handler - directly controls the I/O to and from the device.***

Notes: Device Driver Roles

The Device Head Role

Device driver routines performing the device head role are responsible for fielding device driver requests generated by base level code.

Block and character device head routines have their entry points installed in the device switch table. Examples of these routines are the device driver routines **ddconfig, ddopen, ddclose, ddread, ddwrite, ddioctl, ddmpx,** and **ddrevoke**. User applications can use file system calls in conjunction with special files to access these routines, while kernel extensions can use the file system services available in the kernel (the Logical File **fp_xxx** services).

Device head routines are responsible for the following functions:

Converting the request from the form of the file I/O function call to a form that the routines acting in the corresponding device handler role understand.

Performing the appropriate data blocking and buffering.

Managing the device. This task includes maintaining queues of I/O requests and handling error recovery and error logging.

Device Handler Role

Device driver routines performing the device handler role are responsible for the actual I/O to and from the device. User applications cannot directly access these routines without going through the device head routines. Examples of device handler routines are the **ddstategy** and **dddump** device driver entry points, the interrupt handler, start I/O, and I/O exception handling routines.

Support for some devices can be implemented using two separate device drivers. The first driver acts in the device head role. The second mainly performs the device handler role, but can also have its own set of small device head routines. These routines are registered in the device switch table and are provided primarily to make system configuration and binding easier.

Top Half Considerations

- May be preempted by an interrupt
- May fault on pageable data or code
- Multiprocessor locking required

Notes: Top Half Considerations

Because routines in the top half of a device driver are only called in the process environment, the code and data accessed in this environment is typically pageable. The AIX kernel is designed to allow large portions of kernel code and data to be pageable, thus decreasing the amount of physical memory required by the kernel. This is very important for the kernel since the design philosophy is to create fairly large data structures in pageable virtual memory. These large data structures can then support a wide range of system loads and configurations.

Bottom Half Considerations

- Must not block
- Must not fault on data or code
- Must follow path length guidelines for interrupt level

Notes: Bottom Half Considerations

The bottom half of the device driver typically consists of a routine that starts I/O operations, an interrupt handler, and (optionally) off-level interrupt handling and device time-out routines. The device driver strategy and dump routines are also considered part of the bottom half.

The start I/O routine is typically known only to other routines within the device driver, such as the strategy and interrupt-handling routines. Interrupt-handling routines are registered using kernel services. The dump and strategy routines (the **dddump** and **ddstrategy** entry points) are found in the device switch table.

Some character device drivers, particularly pseudo-device drivers, may have no bottom half if they have no need to run in the interrupt environment. This could be the case when no physical device was being managed.

This bottom half of the device driver runs in both the interrupt handler environment and in the environment of the calling process. Both the code for this part of the device driver and the data it accesses must be pinned so that page faults are not taken in the interrupt execution environment. In addition, routines in the bottom half can use only kernel services that are specified as callable in the interrupt environment.

Kernel Processes

Notes: Kernel Processes

This page intentionally left blank.

Kernel Processes

- Created in the kernel protection domain
- Always execute in the kernel protection domain
- Can be used by:
  - *Device drivers*
  - *Base kernel*
  - *file systems*
  - *Interrupt handlers*
- Can also be used as device managers

Notes: Kernel Processes

Using Kernel Processes:

A kernel process is a process that is created in the kernel protection domain and always executes in the kernel protection domain. Kernel processes can be used in subsystems by complex device drivers, and by the base kernel. They can also be used by interrupt handlers to perform asynchronous processing not available in the interrupt environment. Kernel processes can also be used as device managers where asynchronous I/O and device management is required.

Kernel Process Characteristics

- Created using the creatp and initp kernel services
- Text and data areas come from the kernel heap
- Scheduled like ordinary processes/threads
- Has access to the global kernel address space
  - *Can get access to anything else*
- Must poll for signals and can ignore any signal delivered
  - *Including SIGKILL*

Notes: Kernel Process Characteristics

Introduction to Kernel Processes:

A kernel process (kproc) exists only in the kernel protection domain and differs from a user process in the following ways:

- It is created using the creatp and initp kernel services.
- It executes only within the kernel protection domain and has all security privileges.
- It can call a restricted set of system calls and all applicable kernel services.
- It has access to the global kernel address space (including the kernel pinned and pageable heaps), kernel code, and static data areas.
- It must poll for signals and can choose to ignore any signal delivered, including a kill signal.
- Its text and data areas come from the global kernel heap.
- It has a process-private region containing only the u-block (user block) structure and stack.
- Its parent process is the process that issued the creatp kernel service to create the process.

Kernel Process Characteristics

- **Can call a restricted set of system calls**
  - ***Much larger set than that for user processes running in kernel extension code***
  - ***See info topic "System Calls Available to Kernel Extensions"***
- **Inherits most environment from parent process**
  - ***No root or current directory***
    - ***All file access must specify absolute path names***

Notes: Kernel Process Characteristics

This page intentionally left blank.

Accessing Data from a Kernel Process

- Kernel process can access data that user processes cannot
- This applies to all kernel data, of which there are three general categories:
  - *The user block (ublock) data structure*
  - *The stack for a kernel thread*
  - *Global kernel memory*

Notes: Accessing Data from a Kernel Process

This page intentionally left blank.

Accessing Data from a Kernel Process

- **ublock structure** (@ 0x2FF3B400)
  - ***Exists for kernel processes***
  - ***Contains roughly the same information as for user-mode processes***
  - ***kernel extensions must use kernel services to manipulate data from the ublock area***
    - ***Maintain modularity***
    - ***Increase portability of code to other platforms***

Notes: Accessing Data from a Kernel Process

This page intentionally left blank.

Accessing Data from a Kernel Process

- **Stack**
  - ***Each kernel process has a stack called the process stack***
  - ***Stack is used by the process initial thread***
  - ***May reside in global memory or in the process private segment***
    - ***Must not assume stack is globally available***

Notes: Accessing Data from a Kernel Process

The stack for a kernel process:

To ensure binary compatibility with older applications, each kernel process has a stack called the process stack. This stack is used by the process initial thread. The location of the stack for a kernel process is implementation-dependent. This stack can be located in global memory or in the process-private segment of the kernel process. A kernel process must not assume automatically that its stack is located in global memory.

Accessing Data from a Kernel Process

- **Global kernel memory**
  - ***Kernel processes can access any valid memory location within the global kernel address space***
    - ***Because they are in the kernel protection domain***
  - ***A kernel process can also allocate memory from the kernel heaps***
  - ***Unlike user-mode processes, memory that is dynamically allocated by a kernel process is not freed automatically upon process exit***

Notes: Accessing Data from a Kernel Process

This page intentionally left blank.

Kernel Process Creation, Execution

- Created by the *creatp()* kernel service
- Initialized by *initp()* routine
  - *Made dispatchable*
- Process created is a single threaded kernel process
- Process begins execution at the internal function, *procentry()*, which branches to the routine designated by the caller of *initp()*
- Parent process is the one that called *creatp()* and *initp()*

Notes: Kernel Process Creation, Execution

Kernel Process Creation, Execution:

Kernel processes are created in two steps. First the kernel creates and initializes a new proc structure and creates a new Process Private Segment (**creatp()**). In the second stage, a thread is created, initialized, and scheduled for the processor (**initp()**). This will create a single threaded kernel process. The new process begins execution at the internal function **procentry()**, which branches to the routine designated by the caller of **initp()**. If the thread returns from this entry point, then the function **thread_terminate()** is called to terminate the thread. The return code from the designated entry point is stored in the **p_xstat** field in the calling process. When the last thread in the process is terminated, the process is removed. The usual exit semantic is followed.

Creation of additional kernel threads:

Kernel threads are added to either user or kernel processes in three stages. The **thread_create()** system call creates a new thread within the calling process; the call allocates a stack and copies the arguments to the thread; then calls the **kthread_start()** kernel service to initialize and start the new thread. If this thread returns from the designated entry point, then **thread_terminate()** function is called automatically. These threads never run in user-mode.

Process Termination

- The function *kexitx()* is the focal point for process termination. it is called:
  - *By user application via the exit() system call.*
  - *Automatically when the default signal action is to terminate a process.*
  - *Automatically when a kernel process returns from its entry point.*
- In each case, only the calling process is terminated. A process is terminated when all of its threads are terminated.
- A thread that is sleeping in the kernel must be allowed to clean up any state that it has before the process exits.
  - *It cleans up this state and is terminated when it leaves the kernel.*

Notes: Process Termination

This page intentionally left blank.

Kernel Process Termination

- **A thread cannot completely remove itself from the system because it is running on its kernel stack**
  - ***The final cleanup of system resources is performed under the parent, init, or the swapper***
- **When a kernel process exit, the parent process receives the SIGCHLD signal**
  - ***Indicates the end of a child process***

Notes: Kernel Process Termination

This page intentionally left blank.

Kernel Process Termination

- **Sometimes we don't want the parent process to receive the SIGCHLD signal**
  - ***In this case, the kproc can call the setpinit() kernel service to designate again the init process as its parent***
  - ***The init process cleans up the state of all its child processes that have become zombie processes***

Notes: Kernel Process Termination

This page intentionally left blank.

Kernel-only Threads

- **Every kernel process has one**
  - ***Just like regular processes***
- **Additional threads created by an existing thread in the process via the thread_create() kernel service**
  - ***Initializes a new thread structure***
  - ***Leaves thread in the idle state***
- **Thread started via kthread_start() kernel service**
  - ***Thread entry point, a parameter block, and stack address passed as parameters***

Notes: Kernel-only Threads

A kernel process directly controls the kernel threads. Since kernel processes are always in the kernel protection domain, threads within a kernel process are kernel-only threads. Every kernel process has at least one kernel-only thread. If additional threads are needed, they are created via the thread_create() service and started via kthread_start(). Note that these services are very similar in nature to creatp() and initp().

If any thread in a kernel process calls _exit, all threads in the process are terminated. Otherwise, the kernel process hangs around until the last thread returns (or calls thread_terminate()), then it terminates, too.

Kernel-only Threads

- **Kernel-only threads end in three ways:**
  - ***Thread calls thread_terminate() kernel service***
  - ***Thread returns from its entry point***
  - ***Any thread in the process calls _exit()***
- **The ~~last~~ initial thread to return causes the kernel process to terminate**

Notes: Kernel-only Threads

This page intentionally left blank.

Kernel Process Preemption

- Can use *setpri()* or *nice()* to modify execution priority
  - *setpri() fixes process priority forever*
    - *Affects all threads in process*
    - *Scheduler doesn't modify priority*
  - *nice() reduces priority*
- Can use locking kernel services to serialize access to critical data
- This use of locks does not guarantee that the process will not be preempted
  - *It does ensure that another process trying to acquire the lock waits until the kernel process owning the lock has released it*

Notes: Kernel Process Preemption

This page intentionally left blank.

Kernel Process Preemption

- Locks are not sufficient if a kernel routine can access the critical data while executing in the interrupt environment
  - *Taking a lock in the interrupt handle could cause deadlock*
- Serialization with interrupt handlers must be handled by using locking together with interrupt control
  - *The disable_lock() and unlock_enable() kernel services should be used to serialize with interrupt handlers*
- Kernel processes must ensure that maximum path lengths adhere to the specification for interrupt handlers when executing at an interrupt priority more favorable than *INTBASE*

Notes: Kernel Process Preemption

This page intentionally left blank.

Kernel Process Signal and Exception Handling

- Signals are delivered to exactly one thread within the process which has not blocked the signal from delivery.
- If all threads within the target process have blocked the signal from delivery, the signal remains pending on the process until:
  - *a thread unblocks the signal from delivery*
  - *the action associated with the signal is set to ignore by any thread within the process*
- A kernel process can poll for unmasked signals that are waiting to be delivered by calling the *sig_chk()* kernel service

Notes: Kernel Process Signal and Exception Handling

This page intentionally left blank.

Kernel Process Signal and Exception Handling

- The *sig_chk()* service returns the signal number of a pending signal that was not blocked or ignored
  - *The kernel process then uses the signal number to determine which action should be taken*
- The kernel does not automatically call signal handlers for a kernel process as it does for user processes
- A kernel process should use the exception-catching facilities to handle exceptions caused during run time
  - *Unhandled exceptions that occur in kernel mode result in a system crash*

Notes: Kernel Process Signal and Exception Handling

This page intentionally left blank.

Kernel Process Signal and Exception Handling

- Kernel routines can handle exceptions via these kernel services:
  - *setjmpx*
  - *clrjmpx*
  - *longjmpx* ?

Notes: Kernel Process Signal and Exception Handling

This page intentionally left blank.

Kernel Use of System Calls

- System calls made by kernel processes do not result in a change in protection domain since the kernel process is already within the kernel protection domain
- Routines in the kernel are bound by the loader to the system call and not to the system call handler
- When system calls use kernel services to access data passed by reference, the kernel services recognize that the system call is running within a kernel process instead of a user process and correctly handle the data access
- Kernel process must use the *setuerror()* and *getuerror()* kernel services to set and retrieve system call error information

Notes: Kernel Process Use of System Calls

Kernel Process Use of System Calls:

However, error information returned from a kernel process system call must be accessed differently than for a user process. A kernel process must use the **getuerror()** kernel service to retrieve the system call error information normally provided in the **errno** global variable for user-mode processes. In addition, the kernel process can use the **setuerror()** kernel service to set the error information to 0 before calling the system call. The return code from the system call is handled the same for all processes.

Kernel processes can use only a restricted set of the base system calls found in the **syscalls.exp** file.

Serialization and Synchronization

Notes: Serialization and Synchronization

This page intentionally left blank.

Outline

- SMP Environment
- Critical Sections
- Serializing Critical Sections
- Locks
- Simple Lock Structure
- Complex Lock Structure
- Waiting on Locks
- Lock Example

Notes: Outline

This page intentionally left blank.

Outline

- Priority Promotion
- Events
- Waiting on Events

Notes:

This page intentionally left blank.

SMP Environment

- **Funnelled**
  - ***Will only run on the Master Processor (CPU 0)***
  - ***Used to support back-level device drivers***
- **MP Safe**
  - ***Provide correct serialization strategy for SMP environment***
- **MP Efficient**
  - ***MP Safe and design is optimized for efficiency in SMP environment***

Notes: SMP Environment

The device driver model for multi-processors is different from the device driver model of a uniprocessor. However, AIX Version 4 provides compatibility with old AIX Version 3 uniprocessor device drivers. These new features allow the programmer to tell the system (at driver registration time) what type of device driver is being registered. There are three types of device drivers:

Funnelled - Device driver only runs on the *master processor*. Serialization used in a uniprocessor device driver is sufficient for this type of device driver. It is intended to support low-throuhgput devices and provide compatibility with AIX Version 3 device drivers.

MP Safe - This type of device driver runs on any processor. The code for the device driver has been designed to serialize device driver execution via code locking. That is, a single lock is used by the program to ensure that only one instance of the program is actually running at a time. Many times, performance for this type of device driver on a multiprocessor will only equal uniprocessor performance. This type of device driver is intended for medium throughput devices.

MP Efficient - This type of device driver runs on any processor and has been designed to run on a multiprocessor. The device driver serializes access to devices and data. This type of device driver is intended for high-throughput devices.

Another way to look at these three types is: **Funnelled** is old style uniprocessor synchronization that only runs on the master processor, **MP Safe** is old style device driver with added locking to only allow the device driver to run on any single processor at a time, and **MP Efficient** is a redesigned device driver that is optimized to allow the device driver to run on multiple processors simultaneously.

Critical Sections

- A section of code where access must be serialized
- Three types of serialization required:
  - *Thread-thread*
    - *Synchronizes between two threads in the process environment*
  - *Thread-interrupt*
    - *Synchronizes between process and interrupt environment*
  - *Backtracking*
    - *Method of synchronization within the Virtual Memory Manager*

Notes: Critical Sections

A critical section is a section of code that reads or modifies globally accessible data. A classic example is the linked list. While updating a linked list, if two threads of execution are allowed to modify the list at the same time, invalid results may occur. To solve this problem, access is given to one writer at a time. This is defined as *serializing* access to a critical section. The type of critical section is determined by which environment will access the data.

There are three types of critical sections:

Thread-thread

This type of critical section synchronizes two or more threads at the process level. Since we will never access the data in the interrupt environment, a lock is sufficient to serialize access.

Thread-interrupt

The type of critical section synchronizes access between the process environment and the interrupt environment. A good example is synchronizing access to a list of queued buffers between a device driver top half and its bottom half. The top half runs in the process environment. In this environment, the top half can block and take page faults. The bottom half of the device driver runs in the interrupt environment. The bottom half cannot block and must only access pinned data. This type of critical section requires disabling to the interrupt level that the device driver is running on followed by acquiring a lock.

Note that the top half code that uses this serialization method must be pinned also.

Backtracking

Backtracking with careful update is used by the virtual memory manager (VMM) to serialize access to its data structures. This method of serialization is only used within the kernel and is not available to kernel extensions.

Serializing Critical Sections

- **Disabling serializes with interrupt handlers running on the same processor**
- **Locks provide serialization between processors**
- **Example:**

```
extern Simple_lock drv_lock;
int old_pri;

old_pri = disable_lock( INTOFFL3, &drv_lock );

/* critical section */

unlock_enable( old_pri, &drv_lock );
```


Notes: Serializing Critical Sections

Disabling provides serialization between interrupts on *one processor only*. The **disable_lock** routine provides serialization for a thread-interrupt critical section on more than one processor.

In a uniprocessor environment disabling is enough. In fact, in the uniprocessor version of AIX Version 4, the **disable_lock** routine only disables to the requested interrupt prority level. There is no need to also take a lock.

Locks

- **Simple locks**
  - ***Provides exclusive ownership***
  - ***Spin lock at interrupt level***
    - ***Can only be used at interrupt level on MP systems***
  - ***Convertible (spin, then sleep) lock at process level***
  - ***Serializes thread-interrupt (MP only) and thread-thread (UP or MP) critical sections***

Notes: Locks

Locking services are provided as a means of implementing critical sections. The type of service used depends on the type of critical section. For example, providing serialization in a thread-thread critical section can be provided with the **simple_lock** routine. To provide serialization in a thread-interrupt critical section requires the **disable_lock** routine.

Below are the three types of critical sections in AIX Version 4:

Simple Locks

Provide exclusive ownership and are not recursive. These locks are used for serialization among thread and serialization among threads and interrupt handlers.

Example:

```
Simple_lock   my_lock;

simple_lock_init(&my_lock);
simple_lock(&my_lock);
```

Simple Lock Algorithm

If a simple lock is available, the lock is granted to the caller. If the lock is owned by another thread and that thread is not running, the caller is put to sleep waiting on the lock. If the caller is a thread in the interrupt environment, the thread will spin until the lock is free. If the caller is a thread in the process environment, the thread will spin for a certain number of tries, waiting for the lock to become available. If the lock does not become available within that time, the caller is put to sleep waiting on that lock.

Locks

- **Complex lock**
  - ***Provides read/write access***
  - ***Can be customized***
    - ***Convertible lock (blocking)***
    - ***Spin lock (non-blocking)***
  - ***Serializes thread-thread critical sections only***
- **lockl**
  - ***Blocking lock***
  - ***Provided for backward compatibility with AIX Version 3***
  - ***Do not use for new code***

Notes: Locks

Complex Locks

Sleep (blocking) locks that provide read or write access and are recursive on request. Complex locks are used only for serialization at the process level (ie. thread-thread) They allow multiple readers or one writer. They allow recursive use (the holder of the lock can acquire the lock again)..

Example:

```
Complex_lock my_lock;

lock_init(&my_lock, 0 /* sleep type */);
lock_write(&my_lock); /* NEVER use disable_lock() with complex locks!!! */
```

lockl

Sleeping, mutually exclusive locks are provided for compatibility with AIX Version 3. These should not be used for newly written code.

Note that interrupt-interrupt critical sections are not needed (even with multiple processors) because an interrupt handler can only have one outstanding interrupt at a time.

Locks

- **The thread that holds any of these locks is still preemptible**
  - ***Requester sleeps immediately if holder is not currently running***
- **Lock-management routines are specific to the type of processor on which AIX is running**
  - ***Different processors have different locking-related instructions***
  - ***Provides maximum performance for each processor type***

Notes: Locks

This page intentionally left blank.

Locks and Interrupts

- **Must use simple locks to serialize access between process and interrupt environment on SMP systems**
  - ***e.g. Top-half and bottom-half of device drivers***
- **If not on an SMP system, disabling interrupts is sufficient**
- **Kernel provides services to do necessary serialization**
  - ***disable_lock()***
    - ***Disables interrupts***
    - ***Locks a simple lock if running on an SMP kernel***
  - ***unlock_enable()***
    - ***Unlocks the simple lock if running on an SMP kernel***
    - ***Resets interrupt priority***

Notes: Locks and Interrupts

This page intentionally left blank.

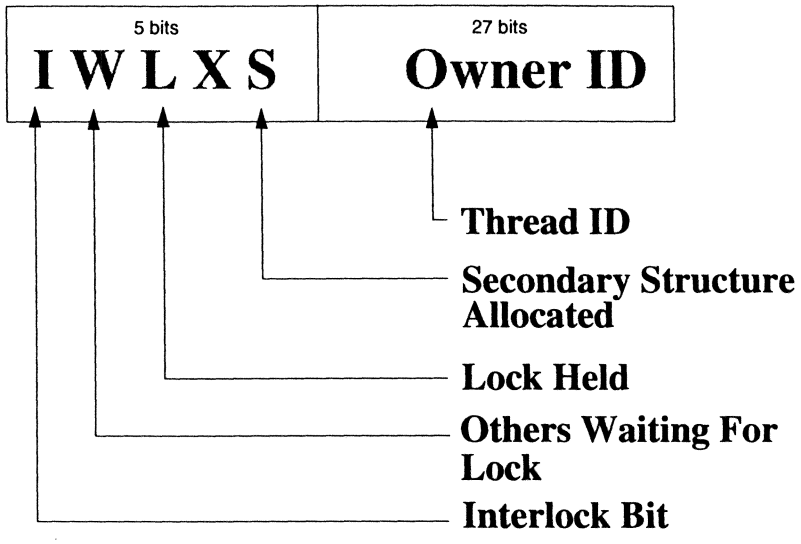
Locks and Interrupts

- Disable lock becomes merely disable on uniprocessor system
- Reduces user-code knowledge of underlying system
  - *Improves maintainability*
  - *Reduces complexity*
- Promotes efficiency on uniprocessor systems
  - *Don't need lock overhead*

Notes: Locks and Interrupts

This page intentionally left blank.

Simple Lock Structure



Notes: Simple Lock Structure

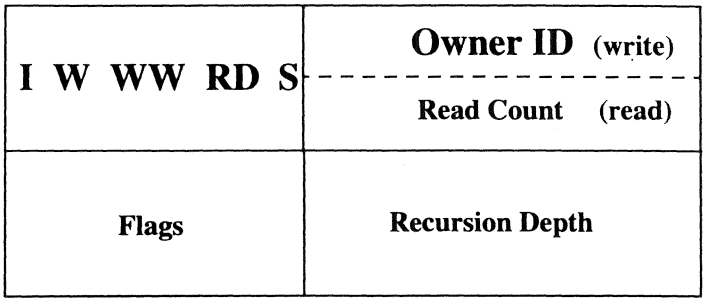
The owner of a simple lock can be determined by examining it's contents:

- A free simple lock is set to SIMPLE_LOCK_AVAIL, which is defined to be the value 0.

-A lock taken will have the thread ID of the lock owner.

The interlock bit in simple (and complex) locks is used to serialize updates to the lock structure itself. Thiis prevents one thread from, for example, going to sleep on a lock at the same time the lock holder is releasing it.

Complex Lock structure



WW - Lock held in write mode or thread waiting for write mode

RD - Lock held in read mode

Flags field holds recursion bit

Recursion Depth indicates how many times "nested"

Eight bytes in length (non-instrumented)

Notes: Complex Lock Structure

The complex lock structure is eight bytes (compared to the simple lock structure, which is four bytes).

If the lock is taken in write mode the lock will contain the thread ID of the lock owner. If the lock is taken in read mode, the lock will contain the number of readers in the lock.

Complex locks can be recursive, but only after the recursive bit is set in the flags field. This bit is turned on by calling the **lock_set_recursive** routine.

Remember that complex locks only serialize threads in the process environment.

Events

- Synchronize occurrence of events in the kernel
- Threads sleep on events
  - *e.g. e_sleep_thread()*
- Implemented as virtual addresses
- Are "well known"
- Are not "owned" by any thread

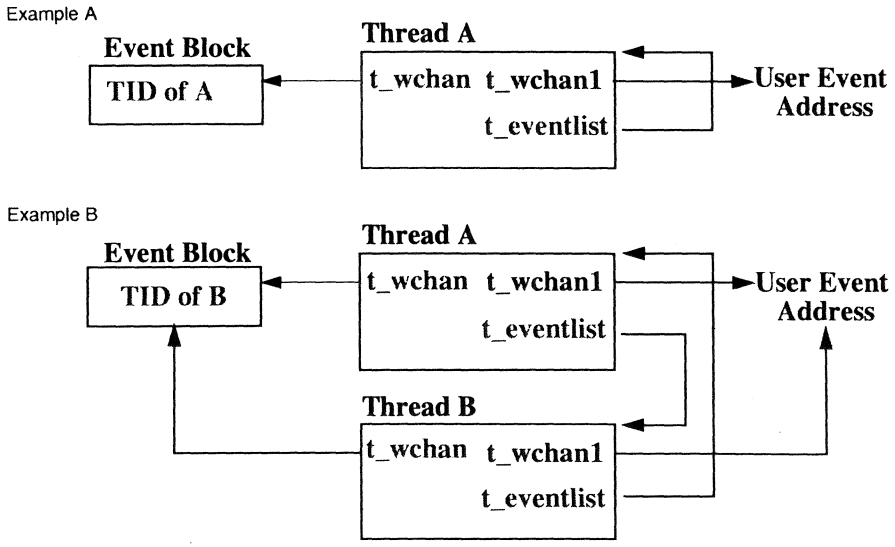
Notes: Events

Threads often must synchronize between the occurrence of "events" within the operating system. A good example is the buffer cache subsystem (note: the **devstrat** routine is used to make read or write requests to block devices). When a thread makes a **devstrat** call within the buffer cache subsystem, it must wait (or synchronize itself) for the event of the reply from the block device driver. Each "buffer object" contains an *event field* that the requesting thread sleeps on. The common programming model in the buffer cache is that after the **devstrat** call the thread calls **biowait**. This causes the thread to *sleep* on the event of the reply for that buffer. When the device driver receives the reply from the block device, it will call **biodone**, which will perform a *wakeup* on that event (or buffer object), waking up the sleeping thread.

So an event is represented by an address within the operating system. Event fields are most often fields within kernel data structures, such as thread and process control blocks. When a thread sleeps, it will provide the event (address) that it expects some other thread to perform a wakeup on in the future. In this respect, event addresses are "well known." It is understood between two or more threads that a particular event is used for a particular purpose (e.g., synchronizing requests and replies on buffer objects).

Also note that no thread or process "owns" an event. Any thread that knows the event address can "post" it (i.e. wake it up), and any number of threads can be waiting on the same event. When the event is posted, all the waiting threads are released.

Waiting on Events



Notes: Waiting on Events

Waiting on events is similar in concept to waiting on locks. The only differences are

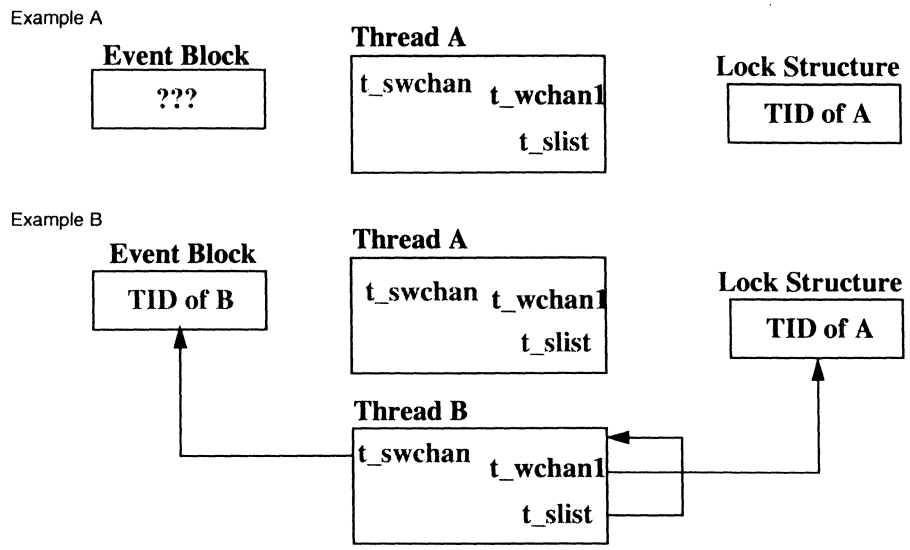
When a thread goes to sleep on an event, the t_wchan field is set to point to an actual event list anchor in some event block. For many of the event-related calls (like e_sleep, e_sleep_thread, etc.), the caller passes an event list anchor as a parameter. However, in some cases (notably the sleep/wakeup kernel services), the caller passes some arbitrary value for the event. In these cases, the caller's value is hashed into a kernel-defined event list. The address of this kernel event list is put into t_wchan, but the original user-specified value is put into t_wchan1.

At any time, anyone can determine the thread at the head of the wait chain by examining the event contents. The chain of waiters is maintained in a circular list via the t_waitlist pointers in the thread structures.

In example A, thread A has gone to sleep waiting for an event. Since thread A is the last (at this point) to wait on this event, its TID goes into the event structure. In example B, thread B is now also sleeping waiting for the same event, so now its TID goes into the event structure. When some other thread posts this event, the system will use the TID in the event to find that thread B needs to be awakened. When it gets to thread B, it finds that A also needs to be awakened (by virtue of the t_eventlist pointer).

If we assume that these threads were using sleep and wakeup, then both of them passed in the same user event address. AIX then hashed this value to pick an event structure to use. When some other thread issues a wakeup for the same address, it gets hashed to the same event, so AIX runs the t_eventlist pointers. For each thread whose t_wchan1 field matches the value passed to wakeup, AIX wakes it up.

Waiting on Locks



Notes: Waiting on Locks

Events are used to implement lock waiting. The only differences between waiting for a lock and waiting for an event are (a) the field names used within the thread block, and (b) one thread "owns" a lock - only the owning thread can release a lock and thus allow the waiting threads to run.

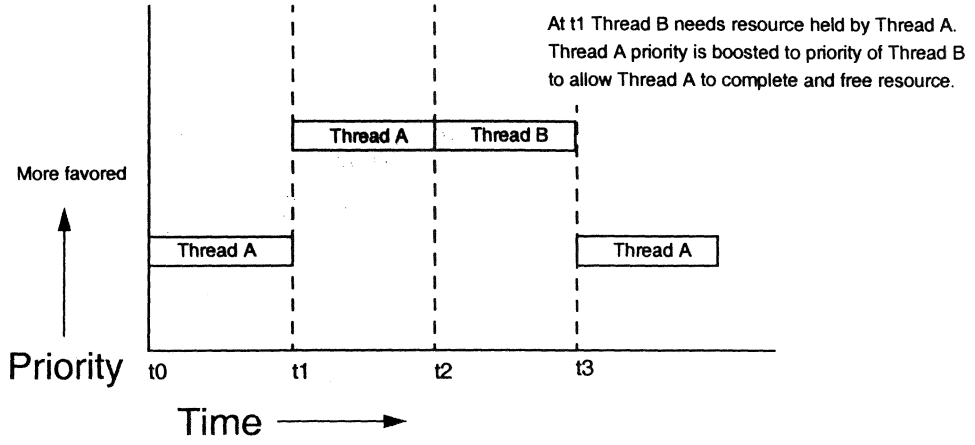
The actual event mechanism is very similar to the sleep/wakeup mechanism. Instead of the user passing AIX some arbitrary "well-known" address, the user calls a lock routine passing the address of a lock structure. This is not an event list address, so AIX hashes the lock address into a different event list (maintained solely for locks). The hashed event list address goes into t_swchan (instead of t_wchan), but the lock address goes into t_wchan1, just as the user's arbitrary event address went there for sleep/wakeup. Also, the circular chain of threads waiting for this lock is maintained by t_slist (instead of t_waitlist).

Example A shows data structure values when a thread owns a simple or complex lock and has no waitors. There is no event list entry for this lock, since no one is currently waiting on it, but the lock structure itself holds the TID of the owning thread. Example B shows data structure values when a thread owns a lock and another thread is waiting on it. There is now an event list entry that refers to thread B, and thread B's t_swchan1 field points to the lock structure. Note that this allows you to find the holder of the lock (by examining the lock contents) no matter how long the chain of waitors is. The t_slist field in the thread data structure allows you to find out all the threads that are currently waiting for this lock.

Note that while holding the lock the priority of Thread A will have the value of the most favored (lowest value) thread of the waiting threads. This is called priority promotion and will be discussed next.

Priority Promotion

- **Allows low-priority resources to make progress when holding kernel resources**



Notes: Priority Promotion

Priority Promotion

Priority promotion happens when two threads of different priorities are competing for the same resource.

Suppose a low-priority thread (Thread A) takes a lock around a key kernel data structure. An example of a key kernel data structure is the **proc_int_lock**. This lock is needed to add and remove entries in the global thread table. since there is only one **proc_int_lock** in the system, contention for this data structure is high. Therefore, threads need to hold it for as short a time as possible.

Now suppose that as Thread A is making modifications to the thread table a thread with a more-favored priority, Thread B, wishes to access the thread table. In order to allow all threads to make progress in the system, Thread A's priority is promoted (or boosted) allowing it to finish its job as quickly as possible.

This scheme allows threads with less-favored priorities access critical system resources and still make progress in the system without blocking high priority threads.

Note that priority promotion take precedence over scheduling policies. Hence, a fixed-priority thread may temporarily acquire a more favored priority than its fixed-priority value.

Scheduling Policies

AIX Version 4.1 supports three types of scheduling as required by Posix: SCHED_OTHER, SCHED_RR, and SCHED_FIFO. Both SCHED_RR and SCHED_FIFO are fixed-priority scheduling policies; SCHED_OTHER is the "traditional" scheduling policy that decays a thread's priority over time and CPU usage.

Accessing Kernel Memory

Notes: Accessing Kernel Memory

This page intentionally left blank.

Outline

- **Accessing Kernel Memory**
  - ***Kernel Symbols***
  - ***Structures in Memory***
  - ***Functions in Memory***
  - ***Kernel Memory Map***

Notes: Outline

This page intentionally left blank.

Kernel Symbols

- **Finding a symbol**
  - ***Given an address, the symbol closest to it can be found with the*** ds ***crash subcommand:***

```
> ds 000d3e0c
        .open + 0x00000004
```

- **Finding an address**
  - ***Given a symbol, its address can be found with the*** nm ***crash subcommand:***

```
> nm open
        000D3E08            PR LD  .open
        0010B18C  00000C  DS DS    open
```


Notes: Kernel Symbols

Finding a Symbol

The **symbol** shown in the foil example is in the kernel segment. This is indicated by the first hexadecimal digit (0), which indicates in what segment the address is located. This shows that the address is at offset 0x4 from the start of the symbol. The symbol starts with a period to indicate a function name.

Finding an Address

In the **address** example, two entries are generated by the **nm** command. One is the address of the symbol table entry and the other refers to the code for the open function as indicated by the storage class **PR**. The first character of the symbol indicates:

. (dot) - The symbol is a kernel function. Example: .open

[(left bracket) - The symbol is in a kernel extension Example: [smt_load:run]

a-z (character a-z) - The symbol refers to some structure or well-known value. Example: kernel_lock

Storage Class of Symbols

The symbol table in the kernel also stores information regarding the type of data the symbol will refer to. The following are typical storage class values of symbols in the kernel:

PR - program code

RW read/write data

TC - read or write general TOC entry

RO - read-only constant

DS - read or write descriptor csect

Functions in Memory

- **Functions contain well-known patterns:**
  - ***Preamble***
  - ***Null Word***
  - ***Function Name***

```
> od  00000374  20  a
00000374:  7c0802a6 93e1fffc 90010008 9421ffc0   ||...........!...|
00000384:  83e20008 387f0000 4800003d 80410014   |....8..H.=.A....|
00000394:  80010048 7c0803a6 30210040 83e1fffc   |...H|...0!.@....|
000003a4:  4e800020 00000000 00002041 80010001   |N.. ..... A....|
000003b4:  00000034 00076d79 5f66756e 63000000   |...4..my_func...|
```


Notes: Functions in Memory

A program or code loaded into memory has a certain pattern that allows us to find:

- Start of code
- End of code
- Name of function

The C compiler translates C source code into machine instructions that are loaded in the RISC System/6000's memory. The code looks like this in memory:

1. Preamble

   The following assembler instruction or value is located close to the start of the function:

   mfspr r0, 8
   0x7c0802a6

   You can look for the pipe symbol ("|") in the ASCII part of od output.

2. Function code

   This will be the machine instructions for the function.

3. A null word

4. Function name

   As an ASCII string. Note that this only applies to C code.

Functions in Memory

- **Functions can be disassembled using crash:**

```
> od 000D3E08 5 I
000d3e08     stu    r1, 0xffc0 (r1)
000d3e0c   mfspr    r0, 8
000d3e10      ai    r7, r1, 0x38
000d3e14      st    r0, 0x48 (r1)
000d3e18     cal    r6, 0x0 (r0)
```


Notes: Functions in Memory

This page intentionally left blank.

Kernel Memory Map

- **A memory map can be generated using** nm **command:**

  nm -vgx /unix > /tmp/unix.map

  - ***This only good for addresses in the base kernel***
  - ***will not display symbol names loaded with*** **sysconfig()**

Notes: Kernel Memory Map

Symbols beginning with a period (".") are functions.

Some symbols will appear twice; one symbol beginning with a period and one symbol without the period. The symbol without the period is the function descriptor; the other is the actual function.

The storage class and address are also displayed.

NOTE: **nm** will only display symbols that are part of the base kernel. Symbols in kernel extensions will not appear in the output

File Systems

Notes: Filesystems

This page intentionally left blank.

Outline

- Kernel I/O Layers
- Logical File System
  - *Logical File System Components*
- Virtual File System
  - *Virtual Nodes (Vnodes)*
  - *Generic I-nodes (G-nodes)*
- Journaled File System (JFS)
  - *Disk and Incore Inodes*
  - *On-Disk Data Structures*

Notes: Outline

This page intentionally left blank.

File System

- **AIX File system differs from traditional UNIX in two ways:**
  - ***(Almost) all file system I/O is through the pager. Only I/O to the log is driven directly.***
    - ***The AIX management of mapped files is based on having page tables not associated with a process, but with an object.***
    - ***The address space of a process is the concatenation of the objects to which it currently has access.***
    - ***The addressable object is a segment.***
    - ***The AIX physical file system is organized to allow the system to establish addressability to an object efficiently.***
  - ***AIX file system meta-data is reliably updated.***

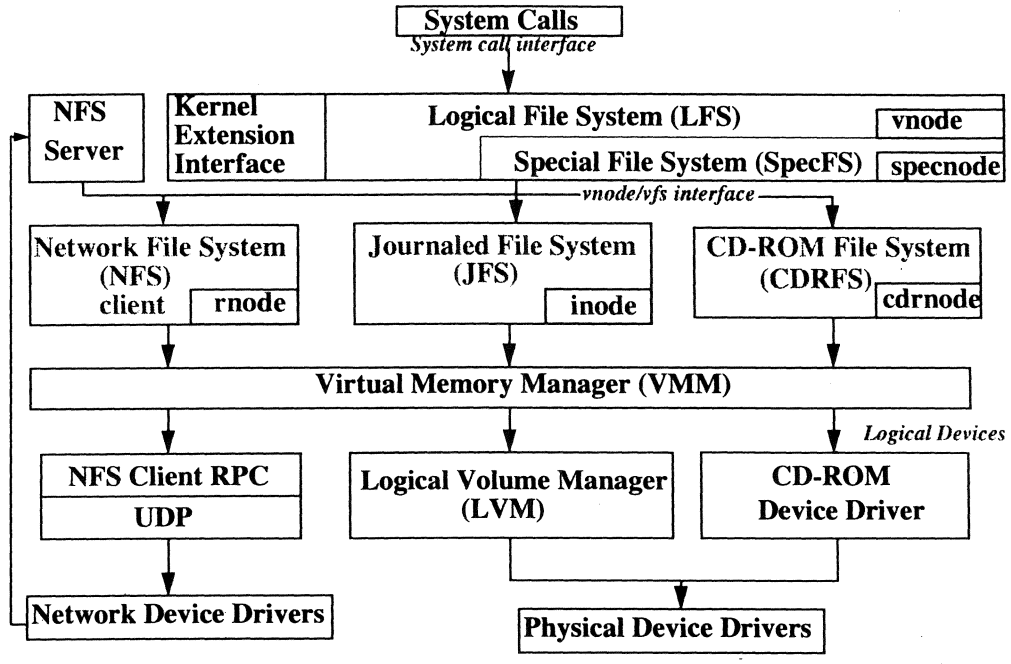
Notes: File System

File System:

The AIX file system is an implementation of the virtual filesystem and vnode operations. The AIX file system differs markedly from traditional UNIX file system implementations in two ways:

- (Almost) all file system I/O is through the pager. This specifically includes the file system data structures: inodes, indirect blocks and directories. Only I/O to the log is driven directly. The AIX management of mapped files is based on having page tables associated not with a process, but with an object. The address space of a process is the concatenation of the objects to which it currently has access. The addressable object is a segment. The AIX physical file system is organized to allow the system to establish addressability to an object efficiently.
- The AIX file system data structures are reliably updated. This is done by writing a record of each operation to a log specified with the volume group containing the logical volume on which the file resides.

File System Environment



Notes: File System Environment

This page intentionally left blank.

AIX File System Environment

- JFS is only one piece of AIX file system design.
- JFS provides its functionality to applications indirectly through the Logical File System (LFS).
  - *The CD-ROM file system and NFS do the same.*
- The LFS provides a consistent programming interface to applications via the system call interface.
  - *The LFS breaks down each system call into requests for the underlying file system implementations.*
- The Interface between the LFS and the underlying file system implementations is referred to as the vnode/vfs interface.

Notes: AIX File System Environment

The JFS is actually only a single piece of the AIX file system design. The JFS is one of several filesystem implementations available with AIX. The JFS provides its functionality to applications indirectly through the Logical File System (LFS). The logical File System provides a consistent programming interface to applications via the system call interface. The logical file system breaks down each system call into requests for the underlying file system implementations. Some system calls are implemented with a single request, for example the read system call, while others may require multiple requests, such as the open system call.

The interface between the logical file system and the underlying file system implementations is referred to as the vnode/vfs interface. Vnodes and VFSs are the primary structures used to communicate through the interface. A vnode represents a file; a vfs represents a mounted file system. This structure was created by Sun MicroSystems and has evolved into a de facto industry standard, thanks in part to NFS. The interface provides a logical boundary between generic objects understood at the logical file system layer and the file system specific objects that the underlying file system implementation must manage such as inodes and super blocks. The logical file system is relatively unaware of the underlying file system data structures since they can be radically different for the various file system types.

The vnode/vfs interface is an opaque interface. A vnode defines a set of operations that may be performed on the vnode such as read or open. Likewise, a VFS structure defines a set of operations that may be performed on a filesystem. No other operations are allowed. This object oriented interface is similar to the features of object oriented programming languages such as C++.

With this structure, AIX can provide a number of different file system types transparently to application programs. These file system reside below the LFS and operate relatively independently of each other. In addition to the JFS, AIX also provides the industry standard Network File System (NFS), and a CD-ROM File system which provides a file level access to CD-ROMs in ISO-9660 format as well as de facto standard formats High Sierra and Rock Ridge.

AIX File System Environment

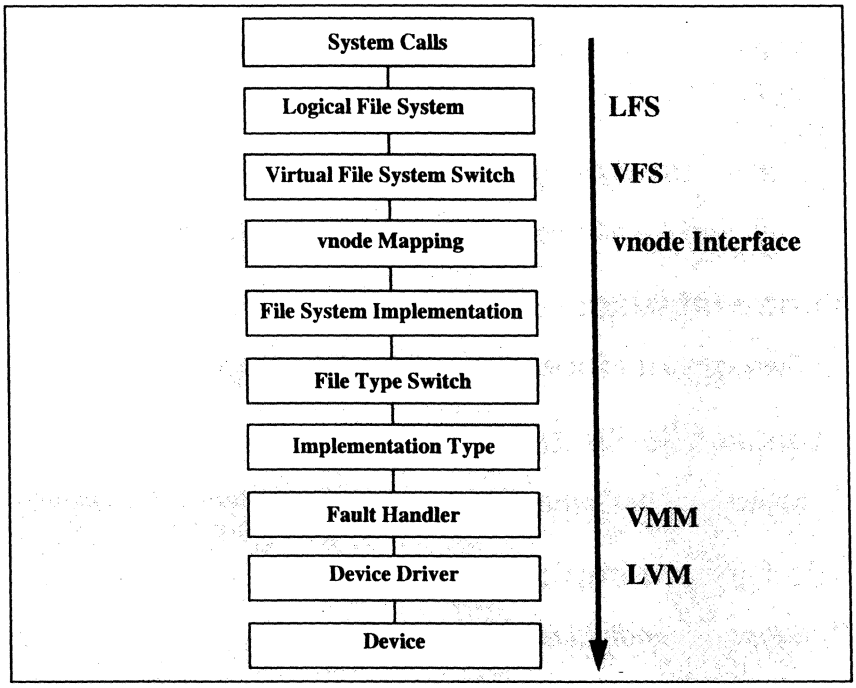
- vnodes and VFSs are the primary structures used to communicate through the interface
  - *A vnode represents a file*
  - *A VFS represents a mounted file system*
- This is a de facto industry standard interface
  - *Created by SUN to support NFS*
  - *Opaque*
    - *vnodes and VFSs define the operations they support*

Notes: AIX File System Environment

This page intentionally left blank.

Kernel I/O Layers

Figure 13



Notes: Kernel I/O Layers

This page intentionally left blank.

Kernel I/O Layers

- Access to files and directories is controlled by various layers in the AIX kernel.
- Virtual File System (VFS).
  - *Standard set of operations on entire file system.*
- vnode Interface.
  - *Standard set of operations within file system.*
- Logical File System (LFS).
  - *Standard set of operations to support system call interface.*
- File System Implementation.
  - *Support of individual file system layout.*

Notes: Kernel I/O Layers

This page intentionally left blank.

Kernel I/O Layers

- **File Type Switch.**
  - ***Layer in some file system implementations that has different branches for the file type (regular file, directory, pipe, etc.)***
- **Fault Handler.**
  - ***Device fault handler support.***
- **Device Driver.**
  - ***Actual device driver code to interface with the device. It is invoked by the page fault handler when the file system implementation code maps the opened file to kernel memory and reads the mapped memory. LVM is the device driver for JFS filesystems.***

Notes: Kernel I/O Layers

This page intentionally left blank.

Logical File System (LFS)

- The logical file system is the level of the file system at which users can request file operations by system call, such as open(), close(), read(), etc..
- This level of the file system provides the kernel with a consistent view of what may be multiple physical file systems and multiple file system implementations.
- File system types, whether local or remote, and regardless of implementation, are represented consistently.
- The logical file system also manages the open file table and per-process file descriptors.

Notes: Logical File System (LFS)

This page intentionally left blank.

Logical File System (LFS)

- The table of virtual file systems is located at the logical file system level.
- The logical file system interface supports UNIX-like file system access semantics, but other non-UNIX file systems can also be supported.

Notes: Logical File System (LFS)

This page intentionally left blank.

LFS Abstraction

- The LFS abstraction specifies the set of file system operations that an implementation must include in order to carry out logical file system requests.
- Physical file systems can differ in how they implement these predefined operations, but they must present a uniform interface to the logical file system.
- A single physical file system can appear to the logical file system as one or more separate virtual file systems.

Notes: LFS Abstraction

This page intentionally left blank.

LFS Abstraction

- Virtual file system operations are available at the logical file system level through the virtual file system switch.
- Each file system type has a set of entries in the virtual file system switch.
- Operations or information requested from a file system implementation need be performed only to the extent possible.

Notes: LFS Abstraction

This page intentionally left blank.

LFS Components

- **system calls:**
  - ***Implement services exported to users. System calls that carry out file system requests do the following:***
    - ***Map the user's parameters to a file system object. This requires that the system call component use the vnode (virtual node) component to follow the object's path name. In addition, the system call must resolve a file descriptor or establish implicit (mapped) references using the open file component.***
    - ***Verify that a requested operation is applicable to the type of the specified object.***
    - ***Dispatch a request to the file system implementation to perform operations.***

Notes: LFS Components

This page intentionally left blank.

LFS Components

- **logical file system file routines:**
  - *LFS routines manage open file table entries and per-process file descriptors.*
  - *An open file table entry records the authorization of a process's access to a file system object.*
  - *A user can refer to an open file table entry through a file descriptor or by accessing the virtual memory to which the file was mapped.*
  - *The logical file system kernel services, such as* fp_ioctl() *and* fp_select(), *begin with the prefix "fp_".*
- **vnodes:**
  - *Vnodes provide system calls with a mechanism for local name resolution.*
  - *Local name resolution allows the logical file system to access multiple file system implementations through a uniform name space.*

Notes: LFS Components

This page intentionally left blank.

Virtual File System (VFS)

- A virtual file system represents a specific file system mount operation.
- A virtual file system can also be viewed as a subset of the logical file system tree, that part belonging to a single file system implementation.
- A virtual file system mount point grafts a virtual file system subtree onto the logical file system tree.
- This mount point ties together a mounted-over vnode (virtual node) and the root of the virtual file system subtree.
- A mounted-over vnode points to a virtual file system, and the mounted VFS points to the vnode it is mounted over.

Notes: Virtual File System (VFS)

This page intentionally left blank.

Virtual Nodes (vnodes)

- A vnode represents access to an object within a virtual file system.
- Vnodes are either created or used again for every reference made to a file by a path name.
- Every path name known to the logical file system can be associated with, at most, one file system object.
- Each file system object can have several names because it can be mounted in different locations.
- Symbolic links and hard links to an object always get the same vnode if accessed through the same mount point.

Notes: Virtual Nodes (vnodes)

Virtual Nodes (vnodes):

A virtual node (vnode) represents access to an object within a virtual file system. vnodes are used only to translate a path name into a generic node (gnode).

A vnode is either created or used again for every reference made to a file by path name. When a user attempts to open or create a file, if the VFS containing the file already has a vnode representing that file, a use count in the vnode is incremented and the existing vnode is used. Otherwise, a new vnode is created. Every path name known to the logical file system can be associated with, at most, one file system object.However, each file system object can have several names. Multiple names appear in the following cases:

The object can appear in multiple virtual file systems. This can happen if the object (or an ancestor) is mounted in a different virtual file systems using a local file-over-file or directory-over-directory mount.

The object does not have a unique name within the virtual file system. (The file system implementation can provide synonyms. For example, the use of links causes files to have more than one name. Note, however, that opens of synonymous paths do not cause multiple vnodes to be created.)

Generic Inodes (gnodes)

- Gnodes represent an object in a file system implementation.
- There is a one-to-one correspondence between a gnode and an object in a file system implementation.
- Serves as the interface between the logical file system and the file system implementation.

Notes: Generic inodes (gnodes)

Generic inodes (gnodes):

A generic inode (gnode) is the representation of an object in a file system implementation. There is a one-to-one correspondence between a gnode and an object in a file system implementation. Each gnode represents an object owned by the file system implementation.

Each file system implementation is responsible for allocating and destroying gnodes. The gnode then serves as the interface between the logical file system and the file system implementation. Calls to the file system implementation serve as requests to perform an operation on a specific gnode.A gnode is needed, in addition to the file system inode, because some file system implementations may not include the concept of an inode. Thus the gnode structure substitutes for whatever structure the file system implementation may have used to uniquely identify a file system object. The logical file system relies on the file system implementation to provide valid data for the following fields in the gnode:

gn_type Identifies the type of object represented by the gnode.

gn_ops Identifies the set of operations that can be performed on the object.

File System Types

If file system is ...	then the physical device is mounted as a ...
Local	Physical device on local machine For example, JFS.
Remote	Physical device on remote machine For example, NFS.
Logical	No physical device associated with the filesystem

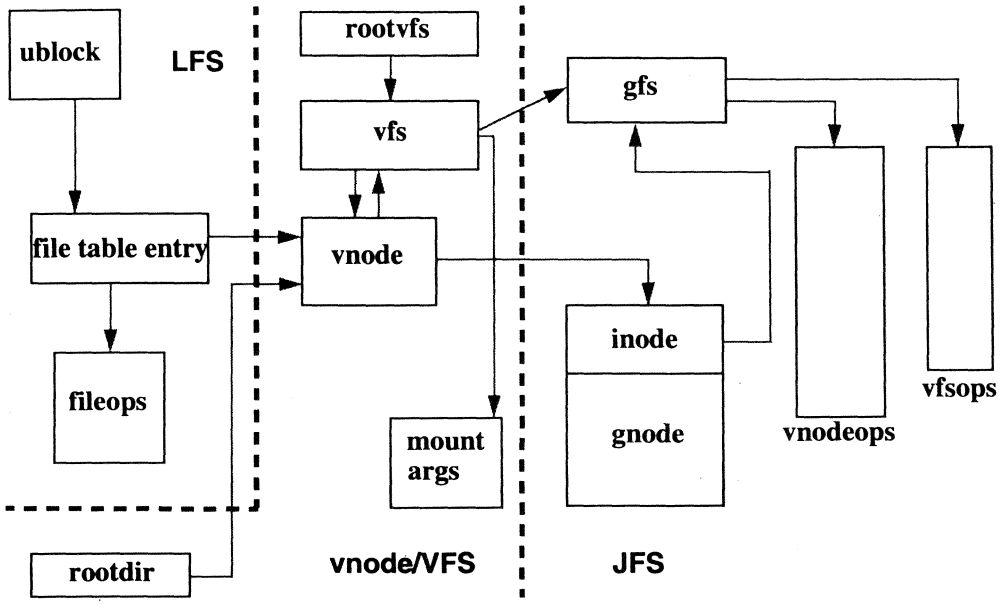
Figure 14

Notes: File System Types

This page intentionally left blank.

Logical File System

The big picture



Notes: Logical File System

This page intentionally left blank.

Logical File System Components

- **The Logical File System (LFS) is made up of many sub-components, they are:**
  - ***User File Descriptor Table:***
    - ***Contained in the User area, is a per process resource***
    - ***Each entry references an open file, device, or socket from the process' perspective***
    - ***Extends beyond first page of the ublock, pageable***
    - ***Up to 32,000 open file descriptor per process***
    - ***Fixed size, not configurable***
    - ***Defined in /usr/include/sys/user.h***
  - ***System File Table:***
    - ***Is global and shared by all processes***
    - ***One unique entry allocated for each unique open of a file, device, or socket in the system***

Notes: Logical File System Components

This page intentionally left blank.

Logical File System Components

- *Each file table entry in the system file table contains a reference count field, various flags and type fields, a read/write pointer and a pointer to another data structure representing the object.*
  - *Large array, partially initialized*
  - *Grows on demand, never shrinks*
  - *Free entries added back to ffreelist*
  - *1,000,000 entries, not configurable*
  - *Defined in /usr/include/sys/file.h*
- *File Operations:*
  - *Pointer to valid operations on the specified file*
  - *Pointers are to the following operations: rw (read/write), ioctl, select, close, fstat*

Notes: Logical File System Components

This page intentionally left blank.

Data Structures Within a Virtual File System (VFS)

- A gfs, vnodeops, and vfsops are created each time a new file system kernel extension is added to the kernel.
- When new file systems are mounted, a vfs and vmount structure are created.
- vmount structure contains specifics of the mount request.
- vfs structure is the connecting structure which links the vnodes (representing files) with the vmount information.

Notes: Data Structures Within a Virtual File Systems (VFS)

Data Structures Within a Virtual File System (VFS):

A **gfs**, **vnodeops**, and **vfsops** structure are created each time a new file system kernel extension is added to the kernel. When new file systems are mounted, a **vfs** and **vmount** structure are created. The vmount structure contains specifics of the mount request, such as the object being mounted, and the stub over which it is being mounted. The **vfs** structure is the connecting structure which links the **vnodes** (representing files) with the vmount information, and the **gfs** structure.

The mount helper creates the vmount structure, and calls the vmount subroutine. The vmount subroutine then creates the vfs structure, partially populates it, and invokes the file system dependent **vfs_mount** subroutine which completes the **vfs** structure, and performs any operations required internally by the particular file system implementation.

Whenever a file is accessed, it is represented by a **vnode** structure, and a **gnode** structure, and possibly a copy of the disk i-node. The vnode and gnode structures are file system independent, and the kernel can access them directly. However, the kernel has no information about the internal storage of the data files that the **vnodes** represent. To perform actions on a data file, the kernel passes a vnode pointer to the relevant **vnode** operation listed in the vnodeops structure.

gfs Structure

- Defined in */usr/include/sys/gfs.h*
- There is one gfs structure for each type of virtual file system currently installed on the machine.
- For each gfs entry, there may be any number of vfs entries.
- The operating system uses the gfs entries as an access point to the virtual file system functions on a type-by-type basis.
- The file system code generally uses the gfs structure as a pointer to the vnodeops structure and the vfsops structure for a particular type of file system.

Notes: gfs Structure

gfs Structure:

The **gfs** structure is defined in the sys/gfs.h header file:

```
struct gfs {
     struct vfsops          *gfs_ops;
     struct vnodeops        *gn_ops;
     int     gfs_type;                  /* type of gfs (from vmount.h) */
     char    gfs_name[16];              /* name of vfs (eg. "jfs","nfs", .)*/
     int     (*gfs_init)();             /* ( gfsp ) - if ! NULL, */
                                        /* called once to init gfs */
     int     gfs_flags;                 /* flags for gfs capabilities */
     caddr_t gfs_data;                  /* ptr to gfs's private config data*/
     int          (*gfs_rinit)();
     int          gfs_hold              /* count of mounts of the ... */
}
```

There is one **gfs** structure for each type of virtual file system currently installed on the machine. For each **gfs** entry, there may be any number of **vfs** entries. The **gfs** structures are stored within a global array accessible only by the kernel. The **gfs** entries are inserted with the gfsadd kernel service. Only one gfs entry with a given **gfs_type** can be inserted into the array. Generally, **gfs** entries are added by the **CFG_INIT** section of the configuration code of the file system kernel extension. The **gfs** entries are removed with the **gfsdel** kernel service. This is usually done within the **CFG_TERM** section of the configuration code of the file system kernel extension.

The operating system uses the **gfs** entries as an access point to the virtual file system functions on a type-by-type basis. There is no direct link from a **gfs** entry to all of the **vfs** entries of a particular **gfs** type. The file system code generally uses the **gfs** structure as a pointer to the **vnodeops** structure and the **vfsops** structure for a particular type of file system, although the **gnodes** also contain a pointer to the **vnodeops** structure for their type of file system.

vfs Structure

- Defined in */usr/include/sys/vfs.h* file.
- There is one vfs structure for each file system currently mounted.
- Provides access to the vnodes currently loaded for the file system.
- Provides a path back to the gfs structure and its file system specific subroutines.

Notes: vfs Structure

```
struct vfs {
    struct vfs    *vfs_next;          /* vfs's are a linked list */
    struct gfs    *vfs_gfs;           /* ptr to gfs of vfs */
    struct vnode  *vfs_mntd;          /* pointer to mounted vnode, the root of this vfs */
    struct vnode  *vfs_mntdover;      /* pointer to mounted-over vnode */
    struct vnode  *vfs_vnodes;        /* all vnodes in this vfs */
    int           vfs_count;          /* number of users of this vfs */
    caddr_t       vfs_data;           /* private data area pointer */
    unsigned int  vfs_number;         /* serial number to help distinguish between */
                                      /*  different mounts of the same object */
    int           vfs_bsize;          /* native block size */
    short         vfs_rsvd1;          /* Reserved */
    unsigned short vfs_rsvd2;         /* Reserved */
    struct vmount *vfs_mdata;         /* record of mount arguments */
};
```

There is one **vfs** structure for each file system currently mounted. New **vfs** structures are created with the **vmount** subroutine. This subroutine calls the **vfs_mount** subroutine found within the **vfsops** structure for the particular virtual file system type. The **vfs** entries are removed with the **uvmount** subroutine. This subroutine calls the **vfs_umount** subroutine from the **vfsops** structure for the virtual file system type.The **vfs** structure is central to each mounted file system. It provides access to the vnodes currently loaded for the file system, mount information through the **vfs_mdata** pointer, and provides a path back to the **gfs** structure and its file system specific subroutines through the **vfs_gfs** pointer. The vfs structures are a linked list with the first **vfs** entry addressed by the **rootvfs** variable which is private to the kernel.

The **vfs_mntd** pointer points to the vnode within the file system which generally represents the root directory of the file system. The **vfs_mntdover** pointer points to a vnode within another file system, also usually representing a directory, which indicates where the file system is mounted. In this sense, the **vfs_mntd** pointer corresponds to the object within the vmount structure referenced by the **vfs_mdata** pointer, and the **vfs_mntdover** pointer corresponds to the stub within the vmount structure referenced by the **vfs_mdata** pointer.

vnode Structure

- Defined in */usr/include/sys/vnode.h* file
- *Vnodes* are the primary handles by which the operating system references file.
- Each time an object within a file system is located, a *vnode* for that object is located, or created.
- Provides the link between the *vfs* structure and the *gnode* structure.

Notes: vnode Structure

```
struct vnode {
    ushort  v_flag;             /* see definitions below            */
    ulong   v_count;            /* the use count of this vnode      */
    int     v_vfsgen;           /* generation number for the vfs    */
    Simple_lock v_lock;         /* lock on the structure            */
    struct vfs *v_vfsp;         /* pointer to the vfs of this vnode */
    struct vfs *v_mvfsp;        /* pointer to vfs which was mounted over this */
                                /* vnode; NULL if no mount has occurred */
    struct gnode *v_gnode;      /* ptr to implementation gnode      */
    struct vnode *v_next;       /* ptr to other vnodes that share same gnode */
    struct vnode *v_vfsnext;    /* ptr to next vnode on list off of vfs */
    struct vnode *v_vfsprev;    /* ptr to prev vnode on list off of vfs */
    union v_data {
        void *      _v_socket;      /* vnode associated data */
        struct vnode *  _v_pfsvnode;    /* vnode in pfs for spec */
    } _v_data;
    char *  v_audit;                /* ptr to audit object          */
};
```

Vnodes are the primary handles by which the operating system references files. Most **vnode** operations are passed a pointer to a vnode as their first parameter. **Vnodes** are created by the vfs-specific code when needed, using the **vn_get** kernel service. Vnodes are deleted with the **vn_free** kernel service. **Vnodes** are the result of a path resolution.

Each time an object (file) within a file system is located (even if it is not opened), a **vnode** for that object is located (if already in existence), or created. Naturally, **vnodes** for each directory searched to resolve the path are also created, or referenced. **Vnodes** are also created for files as the files are created.

The **vnode** structure provides the link between the **vfs** structure and the **gnode** structure. There are two **vnodes** for one **gnode** only in the case where is filesystem object is mounted in more than one place (ie. file-over-file and directory-over-

gnode Structure

- Defined in /usr/include/sys/vnode.h file.
- A gnode refers directly to a file, and is usually embedded within a file system implementation specific structure.
- Gnodes are created as needed by file system specific code at the same time as creating implementation specific structures.
- The gnode is a part of the in-core inode in JFS.

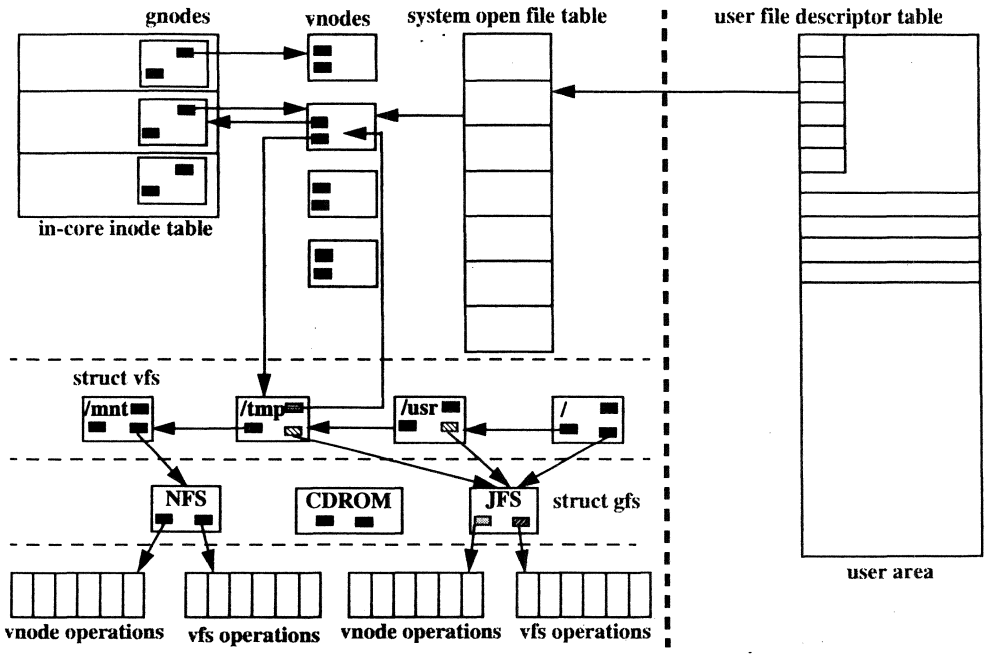
Notes: gnode Structure

```
struct gnode {
     enum vtype gn_type;          /* type of object: VDIR,VREG,... */
     short  gn_flags;             /* attributes of object       */
     ulong  gn_seg;               /* segment into which file is mapped */
     long   gn_mwrcnt;            /* count of map for write     */
     long   gn_mrdcnt;            /* count of map for read      */
     long   gn_rdcnt;             /* total opens for read       */
     long   gn_wrcnt;             /* total opens for write      */
     long   gn_excnt;             /* total opens for exec       */
     long   gn_rshcnt;            /* total opens for read share  */
     struct vnodeops *gn_ops;
     struct vnode *gn_vnode;      /* ptr to list of vnodes per this gnode */
     dev_t  gn_rdev;              /* for devices, their "dev_t" */
     chan_t gn_chan;              /* for devices, their "chan", minor's minor */
     Simple_lock    gn_reclk_lock; /* lock for filocks list      */
     int            gn_reclk_event; /* event list for file locking */
     struct filock  *gn_filocks;  /* locked region list
     caddr_t gn_data;             /* ptr to private data (usually contiguous) */
};
```

A **gnode** refers directly to a file (regular, directory, special, and so on), and is usually embedded within a file system implementation specificstructure. **Gnodes** are created as needed by file system specific code at the same time as creating implementation specific structures. This is normally immediately followed by a call to the **vn_get** kernel service to create a matching **vnode**. The gnode structure is usually deleted either when the file it refers to is deleted, or when the implementation specific structure is being reused for another file.

In early UNIX virtual file system implementations, the vnode represented the file system independent information about a file, and the **in-core inode** was totally file system implementation specific. In AIX, the **gnode** is a part of the **in-core inode** which is uniform, but the remainder of the **in-core inode** remains implementation specific.

LFS Data Structures



Notes: LFS Data Structures

This page intentionally left blank.

Journaled File System (JFS)

Notes: Journaled File System (JFS)

This page intentionally left blank.

Disk inodes

- **Contain information regarding:**
  - ***File type***
  - ***File access times***
  - ***File permission***
  - ***File owner/group***
  - ***Number of links to files***
  - ***Size of file***
  - ***Number of disk blocks in file/directory***

Notes: Disk inodes

This page intentionally left blank.

In-core inodes

- Created in memory when a file is opened.
- Contains the following attributes:
  - *inode locked status*
  - *Open reference count*
  - *Updated disk inode fields representing changes made to file*
- In-core inodes are placed on hash queues with the inode number and the file system number.

Notes: In-core inodes

This page intentionally left blank.

In-core inodes

- **In-core inodes creation:**
  - *When a file is opened, the kernel searches the hash queue to see if there is an in-core inode already associated with the file.*
  - *If an inode is found in the hash queue, the reference count of the in-core inode is incremented and the file descriptor is returned to the user.*
  - *Otherwise, an in-core inode is removed from the free list and the disk inode is copied into the in-core inode.*
  - *The in-core inode is then placed on the hash queue and remains there until the reference count is zero (no processes have the file open).*

Notes: In-core inodes

This page intentionally left blank.

JFS Disk Layout

- **On-disk JFS structures are not too different from traditional file system implementation**
  - ***The superblock is at a well known disk address***
  - ***The remainder of the file system space is devoted to disk inodes and data blocks***
- **The disk inode is the anchor for files**
  - ***One to one correspondence between a disk inode, an i-number, and a file***
  - ***Records file information (size, allocation, owner, etc.)***
  - ***Disjoint from the name - many different names can be refer to the same disk inode***
  - ***Collection of disk inodes can be referred to as the disk inode table***

Notes: JFS Disk Layout

JFS Disk Layout:

The on-disk JFS structures are not all that different from traditional file system implementation. The superblock lives at a well known disk address and records information global to the file system. The remainder of the file system space is devoted to disk inodes and data blocks.

The disk inode is the anchor for files in a UNIX file system. There is a one to one correspondence between a disk inode, an i-number, and a file. The inode records file information such as size, allocation, owner, and so on. However, it is disjoint from the name since many different names can be refer to the same inode via the inode number. The collection of disk inodes can be referred to as the disk inode table.

Unlike the System V.2 file system yet, similar to the BSD file system, file system space is divided into chunks known as allocation groups. Each allocation group contains disk inodes and free blocks. This permits inodes and data blocks to be dispersed throughout the file system and allows file data to lie in closer proximity to its inode. Despite the fact that the inodes are distributed through the disk, a disk inode can be located using a trivial formula based on the i-number and the allocation group information contained in the super block.

The allocation group structure facilitates the growth of the file system. When a new allocation group is logically added to the end of the file system, the file system has more disk inodes and more data blocks. Of course, this assumes some sort of support for increasing the size of the underlying disk object.

Many of the JFS data structures, or meta-data, have a "special" disk inode associated with them and the disk space for that meta-data is allocated from the free block pool just as regular file data. Thus the disk inode refers to the data blocks that contains the file system meta-data. It is convenient to have this type of organization where some particular piece of the file system meta-data can be addressed as the data of some well known file. These "special disk inodes" have reserved disk inode numbers, but have no directory entry associated with them. The notion of representing file system data within the data blocks of some file is similar to a directory. A directory is a disk inode whose data blocks contain file system meta-data. The JFS merely extends this structure to other types of meta-data.

Since allocation groups contain only disk inodes and data blocks, all other file system data structure must be contained within data blocks. The indirect map, the disk block allocation map, and the disk inode allocation map are examples of such structures.

JFS Disk Layout

- **Allocation group structure facilitates growth of the file system**
  - ***When new allocation groups are added to the end of the file system, the file system has more disk inodes and more data blocks***
  - ***Allocation groups contain only disk inodes and data blocks. All other file system data structures must be contained within data blocks***
  - ***The indirect map, the disk block allocation map, and the inode allocation map are examples of such structures***
- **Many of the JFS data structures have "special" inode associated**
  - ***Disk space for those "special" inode, or meta-data, is allocated from the free block pool just as regular file data***
  - ***Have reserved disk inode numbers***
  - ***Have no directory entry associated***

Notes: JFS Disk Layout

JFS Disk Layout (Continued):

The indirect map, often referred to as .indirect, is a "special" disk inode whose data blocks encompasses all of the indirect blocks in the file system. This structure exists only as an implementation convenience when allocating and de-allocating indirect blocks.

The free space in the file system is represented via a bit map referred to as the disk block allocation map. A bit in the map indicates the free status of an associated disk block. As mentioned previously, space for the bit map is allocated from the free disk blocks themselves. When a file system expands, new blocks may have to be added to the disk map so that the new blocks can be represented in the disk map.

The free inode information is likewise represented with a bit map known as the disk inode allocation map. Again, each disk inode in the system has its allocation status recorded by a single bit in the disk inode allocation map.

AIX Physical File System

- **The AIX physical file system is a structure superimposed on a logical volume composed of the following:**
  - ***superblock***
  - ***disk block allocation map***
  - ***disk inodes***
  - ***disk inode allocation map***
  - ***disk inode extensions***
  - ***indirect blocks***
  - ***directory blocks***
  - ***file data blocks***

Notes: AIX Physical File System

AIX Physical File System:

The AIX physical file system is a structure superimposed on a logical volume. The organization of the AIX physical file system is based on the notion of a segment; a segment is a logically contiguous, but physically discontiguous, set of data blocks. Most segments are described by a disk inode; a disk inode is a permanent record of the mapping of the logical blocks of a segment to physical blocks. It has the following components:

- **superblock** (called .superblock) The superblock holds a concise description of the file system: its size allocation information, and an indication of the consistency of on-disk data structures. Updates to the superblock are done by accessing it as a segment.
- **disk block allocation map** (called .diskmap) This bit map indicates whether each block on the logical volume is in use.
- **disk inodes** (called .inodes) Every file system object is described by an disk inode. Each disk inode is a fixed size: 128 bytes. All disk inodes for a file system appear in the disk inode segment, .inodes. This segment is journalled; it is guaranteed to be consistent after replay of the log.
- **disk inode allocation map** (called .inodemap) The allocation map for the disk inode segment is maintained as a bit map in a separate segment.
- **disk inode extensions** (called .inodex) Additional information (e.g., access control lists) which is too large to place in the fixed-size disk inode appears in this segment. This segment is journalled; it is guaranteed to be consistent after replay of the log.
- **indirect blocks** (called .indirect) The most common file system object is a regular file. For a regular file, the inode holds a list of the data blocks which compose the file. It would be impractical to allocate inodes large enough to directly hold this entire list. The list of physical blocks are held in a tree structure, rather than an array. The intermediate nodes of this tree are the indirect blocks. All indirect blocks in a file system are pages in the .indirect segment.
- **directories** All objects are accessed by a name. The namespace is a tree. The root of this tree is described by disk inode 2.

AIX Physical File System

- **Organization in memory is based on the notion of a segment**
  - ***A segment is a logically contiguous, but physically discontiguous, set of data blocks***
  - ***Most segments are described by a disk inode***
- **Disk inodes are permanent records of the mapping of the logical blocks of a segment to physical blocks**
  - ***Disk inodes ==> permanent storage***
  - ***XPT blocks ==> temporary storage***

Notes: AIX Physical File System

AIX Physical File System:

Here you can start to see how AIX exploits the segment architecture to implement its physical file system. Just as virtual memory looks contiguous to a user program but may be scattered about real memory or paging space, disk files are made to look contiguous to the user program even though the physical disk blocks may be very scattered. When AIX needs to create a segment of virtual memory, it creates an External Page Table (XPT), which contains a collection of XPT blocks. When the physical file system creates a file, it creates a disk inode and possibly indirect blocks to describe the file. Disk inodes (and indirect blocks), and XPT blocks make their respective user-level resources appear contiguous.

The JFS maps all file system information into virtual memory, including user data blocks. The read and write operations are much simplified in that they merely initialize the mapping and then copy the data. Likewise, a directory lookup operation merely maps the directory into virtual memory and then goes walking through the directory structure. This greatly simplifies the code by dividing the algorithmic problem of searching directory entries from the task of performing disk I/O operatings and managing a buffer cache.

The I/O function is handled by the Virtual Memory Manager (VMM). When a page fault occurs on a mapped file object, the VMM is able to determine what file is being accessed, examine the inode to determine where the data is, and initiate a page in to transfer the data from the file system into memory. Once completed, the faulting process can be resumed and the operation continues, oblivious to the fact that a memory mapped access caused a disk operation.

AIX Physical File System

- **Each of the above components is described by a disk inode in the AIX physical file system**
  - ***Each directory or user file is a separate segment in memory***

Notes: AIX Physical File System

This page intentionally left blank.

Superblock

- Also known as .superblock
- The superblock holds a concise description of the file system
  - *Size allocation information*
  - *An indication of the consistency of on-disk data structures*
- Updates to the superblock are done by accessing it as a segment

Notes: Superblock

This page intentionally left blank.

Disk Block Allocation Map

- Also known as .diskmap
- This bit map indicates whether each block on the logical volume is in use
- This map is journalled
  - *Should be consistent after replay of the log*

Notes: Disk Block Allocation Map

This page intentionally left blank.

Disk Inodes

- Also known as .inodes
- Every file system object is described by an disk inode
- Each disk inode is a fixed size: 128 bytes
- All disk inodes for a file system appear in the disk inode segment, .inodes
- This segment is journalled
  - *Guaranteed to be consistent after replay of the log*

Notes: Inodes

This page intentionally left blank.

Disk Inode Allocation Map

- Also known as .inodemap
- Allocation map for the disk inode segment
- Maintained as a bit map in a separate segment
- Guaranteed to be consistent after replay of the log

Notes: Inode Allocation Map

This page intentionally left blank.

Disk Inode Extensions

- Also known as .inodex
- Additional data which is too large to place in the fixed-size disk inode appear in this segment
  - *e.g., access control lists*
- This segment is journalled
  - *Guaranteed to be consistent after replay of the log.*

Notes: Inode Extensions

This page intentionally left blank.

Indirect Blocks

- Also known as .indirect
- For a small regular file, the disk inode holds a list of the data blocks which compose the file
  - *Would be impractical to allocate disk inodes large enough to directly hold this entire list for large files*
- List of physical blocks are held in a tree structure, rather than an array
  - *Intermediate nodes of this tree are the indirect blocks*

Notes: Indirect Blocks

This page intentionally left blank.

Indirect Blocks

- **All indirect blocks in a file system are pages in the .indirect segment.**
  - ***A virtual memory segment that is created for lookup of the indirect blocks of all inodes in the filesystem***
- **This segment is journalled**
  - ***Guaranteed to be consistent after replay of the log.***

Notes: Indirect Blocks

This page intentionally left blank.

Directories

- All objects are accessed by a name
- The namespace is a tree. The root of this tree is described by disk inode 2

Notes: Directories

This page intentionally left blank.

AIX File System Structure

- **An empty file system is:**
  - ***Disk block 0 not used***
    - ***Reserved for use by LVM***
  - ***Disk block 1 first block of .superblock***
  - ***Other disk blocks contain .inodes, the maps, the root directory, and indirect blocks***
- **The contents of the first two pages of .inodes are reserved for system inodes**
  - ***For example, the .inodes segment itself is described by a disk inode in this page***
- **The actual on-disk layout for any file system can be viewed with the fsdb command**

Notes: AIX File System Structure

This page intentionally left blank.

AIX File System Structure

- **In an empty file system, .inodes is initialized to:**
  - *disk inode 0 never allocated (disk inode 0 indicates availability)*
    - *This is reserved by setting n_link = 1.*
  - *disk inode 1 .superblock*
  - *disk inode 2 the root directory*
    - *First page of the root directory has entries for ".." (itself) and for each of the other special segments*
  - *disk inode 3 .inodes*
  - *disk inode 7 .inodex*
  - *disk inode 4 .indirect*
  - *disk inode 6 .diskmap*
  - *disk inode 5 .inodemap*

Notes: AIX File System Structure

This page intentionally left blank.

Disk Inode Structure

- **Defined in /usr/include/jfs/ino.h and contains:**
  - ***di_gen*** *The disk inode generation number.*
  - ***di_nlink*** *The number of directory entries which refer to the file.*
  - ***di_mode*** *The file type, access permissions and attributes.*
  - ***di_uid*** *User ID of owner.*
  - ***di_gid*** *Group ID.*
  - ***di_size*** *File size.*
  - ***di_nblocks*** *Number of blocks used by file. This does not include indirect blocks.*
  - ***di_mtime*** *Time at which the contents of the file were last modified.*
  - ***di_atime*** *Time at which the file was last accessed by read.*
  - ***di_ctime*** *Time at which contents of disk inode were last updated.*

Notes: Disk Inode Structure

This page intentionally left blank.

Disk Inode Types

- The private portion of the disk inode depends on its type. The types are defined in /usr/include/sys/mode.h and compose portions of the di_mode field
- Disk inode types are:
  - *S_IFREG Regular file. The format of the private portion of a disk inode for a data file (including some symbolic links and directories) depends on the size of the file. The AIX file system always allocates full blocks to data files.*
  - *S_IFDIR Directory. The private portion of a directory disk inode is identical to that of a regular file*
  - *S_IFBLK Block device. Block device disk inodes have only the dev_t*
  - *S_IFCHAR Character device. Character device disk inodes have only the dev_t*

Notes: Disk Inode Types

This page intentionally left blank.

Disk Inode Types

- *S_IFLNK* *Symbolic link*
- *S_IFSOCK* *A Unix Socket domain*
- *S_IFIFO* *FIFO. A FIFO disk inode has no persistent private data*

Notes: Disk Inode Types

This page intentionally left blank.

Indirection - None, Single, Double

- **No indirect**
  - ***File size <= 8 blocks (32K for normal filesystems, 1M for large file filesystems)***
  - ***Disk inode holds the logical volume block numbers of start of data blocks***
- **Single indirect**
  - ***8 blocks < File size <= 1024 blocks (4M normal filesystems, 128M large file filesystems)***
  - ***Disk inode contains the index (and logical volume block number) of a page in .indirect which points to the data blocks***

Notes: Indirection - None, Double, Single

JFS File Geometry:

A file's geometry is described by one of three paradigms: direct, single indirect, and double indirect.

The disk inode contains 9 slots for disk addresses. The first 8 are used for the direct addressing while the ninth is used for indirect addressing.

Indirection: None, Single, Double:

- **No indirect:** If the file is less than or equal to 32K (1M for large file filesystems) then the file will use direct addressing. The inode points directly to the data blocks.
- **Single indirect:** If the file is too large for direct addressing, it will use a form of indirect addressing. Single indirect addressing will be used if the file's data blocks can be addressed by a single indirect block. In this case, the direct block addresses are unused and the indirect block address contains the disk address of the one and only indirect block. Each 4 byte entry in the indirect block refers to a data block. Hence single indirect addressing is used for files greater than 32K (1M) yet less than 4MB (128M).
- **Double indirect:** If the file is larger than 4 MB (128M for large file filesystems), then double indirect addressing is used. In this case, the indirect field contains the block number of the double indirect block for the file. The double indirect block contains 512 pointers to the indirect blocks . As before, each indirect block contains 1024 block addresses for the data for the file.

Indirection - None, Single, Double

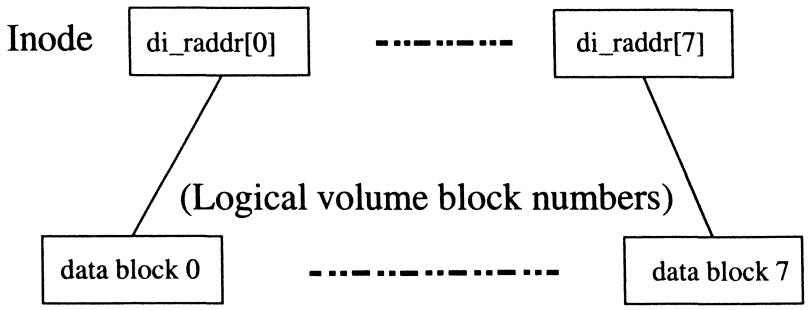
- **Double indirect**
  - ***File size > 1024 blocks (4M normal filesystems, 128M large file filesystems)***
  - ***File size < 512*1024 blocks (2G normal filesystems, 64G large file filesystems)***
  - ***Disk inode has the index (and logical volume block number) of a doubly-indirect page in .indirect***
  - ***When the file grows from singly- to doubly-indirect, the first index in the doubly-indirect block is the old singly indirect root***

Notes: Indirection - None, Single, Double

This page intentionally left blank.

No Indirect Blocks

Inode Disk Addresses for Very Small Size File

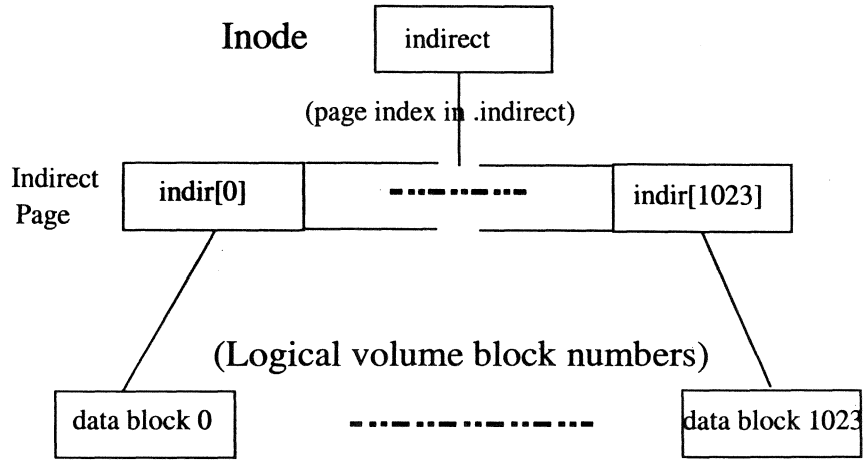


Notes: No Indirect Blocks

This page intentionally left blank.

Single Indirect

Inode Disk Addresses for Intermediate Size File

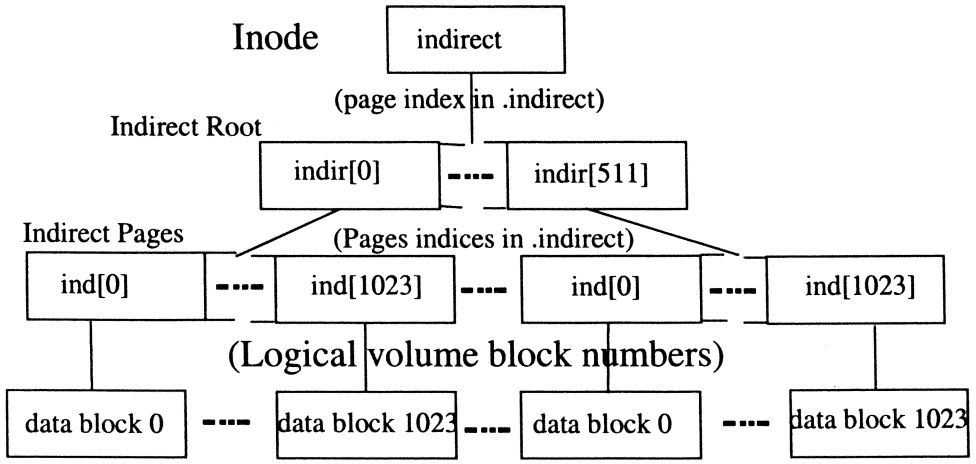


Notes: Single Indirect

This page intentionally left blank.

Double Indirect

Inode Disk Addresses for Large Size File



Notes: Double Indirect

This page intentionally left blank.

In-core Inodes

- Each in-core inode represents each currently-accessed object in AIX file system
- The in-core inode structure is defined in /usr/include/jfs/inode.h

Notes: In-core Inodes

This page intentionally left blank.

In-core Inode Format

- Two parts to each in-core inode:
  - *First portion of data structure relevant only while the object is accessed.*
  - *The last 128 bytes is copy of disk inode*
- The in-core inodes includes
  - *An exclusive-use lock*
    - *Must be held before the in-core inode is updated*
    - *Actually implemented with a simple lock*
  - *Use count*
    - *The in-core inode cannot be destroyed while it has a non-zero use count*
  - *Open counts*
    - *Separate reader and writer counts are maintained in the gnode in the in-core inode*

Notes: In-core Inode Format

This page intentionally left blank.

In-core Inode Format

- *Are incremented at each open, and decremented at close*
- *A process which has opened the file for both reading and writing is counted as both a reader and writer*

- *State flags*
  - *Maintain miscellaneous in-core inode state*
- *Exclusion counts*
  - *A bit indicates that the file has been opened for exclusive access*
  - *A separate count of the number of readers who have specified read-only sharing (precluded writers) is also maintained*
  - *If a process attempts to open the inode with a mode which conflicts with the current open status, it can be placed on a wait list for the inode (if the O_DELAY open flag was specified)*

Notes: In-core Inode Format

This page intentionally left blank.

In-core Inode Format

- *Hash table links*
  - *All existing in-core inodes are kept in a hash table, accessed by device and index*
  - *Allows finding an inode by file handle, and assures that multiple inodes are not created for the same object*
- *Free list links*
  - *All unused in-core inodes are kept in a free list*
- *Mount table entry*
  - *If an object is in use, its underlying device must currently be mounted*
  - *Each in-core inode points back to its mount table entry to avoid searching the mount table to find the entry for this object*

Notes: In-core Inode Format

This page intentionally left blank.

In-core Inode Management

- **Three states for every in-core inode**
  - *Active - there is currently a vnode that refers to this inode*
    - *Some process has the corresponding file open*
  - *Cached - there is no vnode that refers to this inode*
    - *Corresponding file is not open anywhere in system*
    - *The data the in-core inode and associated segment holds is still valid and may be reused if the inode is reopened (avoids extra disk I/O)*
  - *Free - structure is available for immediate use*
- **Cached in-core inodes can be reacquired at any time**

Notes: In-core Inode Management

This page intentionally left blank.

In-core Inode Table

- Active in-core inodes are maintained in the in-core inode table
- Entries are accessed by iget(), if an entry is not already in the table, iget() will call iread() to obtain the entry
- Entries are marked as unused by iput()

Notes: Inode Table

Inode Table:

Active in-core inodes are maintained in the inode table, a hash table of in-core inodes for recently accessed files.

Entries are accessed by iget(). If an entry is not already in the table, iget() will call iread to obtain the entry.

Entries are marked as unused by iput(). If an inode is iput(), it no longer has other references to it, and it has a zero link count, then it is placed on the free list. If an inode is iput() and has no references to it, but still has a non-zero link count, then it is placed in the cache list. From this list, it can be easily reacquired should a process need the inode again.

Inode Locking

- Locking serializes operations on the inode
- For all operations which require locking more than one inode, the inodes are locked in descending order
  - *i.e. highest inode number is locked first*
  - *Prevents deadlock conditions*

Notes: Inode Locking

Inode Locking:

The AIX file system serializes operations by obtaining an exclusive lock on each inode involved in the operation.

For all operations which require locking more than one inode, all involved inodes are known at the start of the operation. The ilocklist() routine sorts these into a canonical order before locking.

Note: The iget() routine does not return a locked inode; nor does iput() free any lock on the inode.

Loader

Notes: Loader

This page intentionally left blank.

Outline

- XCOFF
- Linkage Conventions
  - *Prologs and Epilogs*
  - *Stack Frames*
- Function Calls
  - *Function Descriptors*
  - *Table of Contents (TOC)*
  - *glink*
- System Calls
  - *System Call Handler*

Notes: Outline

This page intentionally left blank.

XCOFF

- **eXtended Common Object File Format (XCOFF) is the AIX object file format**
  - *COFF is the traditional UNIX object file format*
  - *XCOFF adds support for shared libraries and dynamic loading to COFF*
- **There are two XCOFF formats, 32-bit and 64-bit**
  - *only objects of the same format can be linked together into a module*
  - *32-bit XCOFF modules can only be executed in 32-bit mode*
  - *64-bit XCOFF modules can only be executed in 64-bit mode*
  - *only 32-bit XCOFF can be loaded into the kernel as a kernel extension*

Notes: XCOFF

This page intentionally left blank.

XCOFF

- **There are two types of XCOFF files**
  - ***object files***
    - ***created by compilation or assembly of source files***
    - ***can be linked to create a module***
    - ***can be used as linkable objects in libraries (non-shared)***
  - ***modules***
    - ***created by linking object files***
    - ***can be executed if imports are satisfied and has a*** start ***entry point***
    - ***can be shared modules in libraries***
    - ***can be dynamically loaded into an application***
    - ***can be loaded as a kernel extension***

Notes: XCOFF

This page intentionally left blank.

Linkage Conventions

- **General agreement on the machine state when calling and returning from subroutines**
  - ***Ensures that subroutines are generally callable, regardless of language or compilation unit***
- **Specifically covers register and memory usage**
  - ***Allows parameter passing and return values to be in***
    - ***General Purpose Registers (GPRs)***
    - ***Floating Point Registers (FPRs)***
    - ***or both***

Notes: Linkage Conventions

This page intentionally left blank.

Linkage Conventions

- **Volatile vs. Non-volatile**
  - ***Registers that can be used as scratch registers and which are not saved across function calls are "volatile"***
  - ***Registers that must be saved and restored are "non-volatile"***

Notes: Linkage Conventions

This page intentionally left blank.

Linkage Conventions

- **General Purpose Registers**
  - ***R0 - used in function prologs, volatile***
  - ***R1 - the stack pointer, non-volatile***
  - ***R2 - the Table-Of-Contents (TOC) register, non-volatile***
  - ***R3-R10 - first through eighth words of argument list for callee, volatile***
    - ***R3-R4 - first and second words of return value***
  - ***R11&R12 - special usages, volatile***
  - ***R13-R31 - general usage, non-volatile***

Notes: Linkage Conventions

This page intentionally left blank.

Linkage Conventions

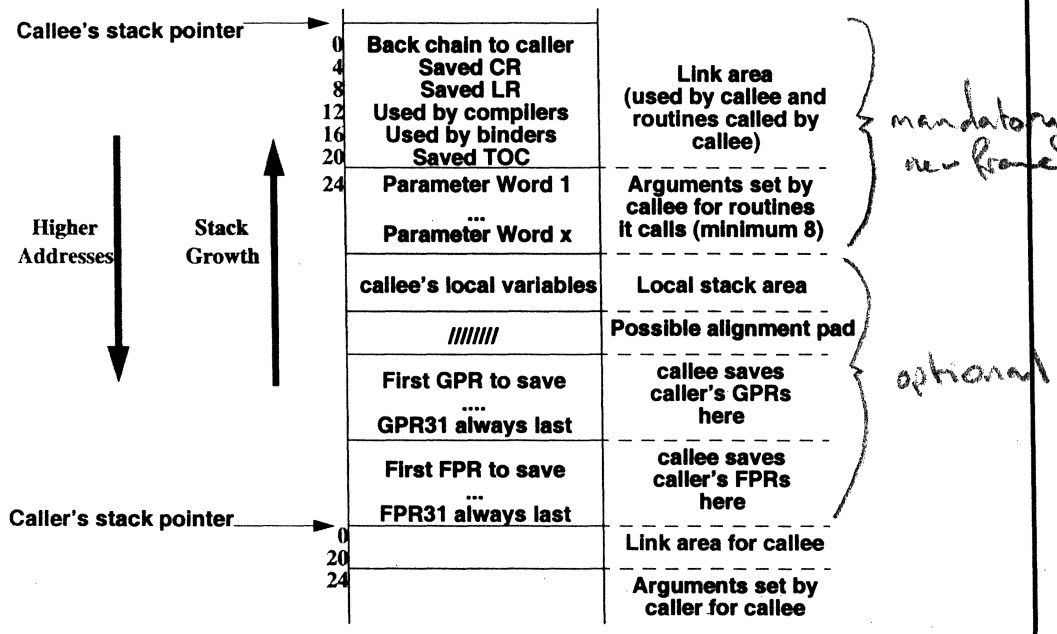
- **Floating Point Registers**
  - *FPR0 - scratch, volatile*
  - *FPR1-FPR13 - first through thirteenth floating-point parameters to callee, volatile*
    - *FPR1-FPR4 - first through fourth 8 bytes of return value*
  - *FPR14-FPR31 - general use, non-volatile*
- **Condition Registers**
  - *CR0&CR1 - volatile*
  - *CR2-CR4 - non-volatile*
  - *CR5-CR7 - volatile*

Notes: Linkage Conventions

This page intentionally left blank.

Linkage Conventions

- **Stack format:**



Notes: Linkage Conventions

AIX defines a set of linkage conventions to help ensure that routines can call other routines, regardless of the language in which they were written, or the compilation unit of which they are a part. They define what the machine state should be on entry to and exit from a routine. The major areas affected are registers and memory.

There are two classes of registers defined by the linkage conventions. Volatile registers are free for use at any time, by any routine. They are not saved or restored across routine calls, so their contents cannot be counted upon after a call has been made. Non-volatile registers must be saved and restored always.

The stack is the key memory area for state saving and restoring. When a routine is called, it must save some information in the link area of the caller, save any necessary GPRs and FPRs immediately below the current stack pointer, then save the stack pointer in its new link area, and finally update the stack pointer to point to its new link area. It can then pick up the first 8 parameters from the registers, and any additional parameters from the parameter word stack area of the caller. Note that parameter space is always reserved for 8 parameters, even though they may not all be used, and even though they are in registers. This is because the called routine needs to be able to take the address of passed parameters. If it does, then the called routine needs to store the appropriate register into the appropriate location on the stack. That location can then be used in address-of calculations.

The management of the stack will become more clear when we look at prologs and epilogs.

Prologs and Epilogs

- **Prolog:**
  - *The set of instructions at the beginning of a routine that perform register saving and stack management*
- **Epilog:**
  - *Instructions that restore registers and stack to their states before the routine was called*
    - *Usually at the end of a routine, but may be elsewhere due to compiler optimizations*
- **Compilers generate prologs and epilogs for all routines**
  - *However, optimization causes the code sequences to look different for different routines*

Notes: Prologs and Epilogs

This page intentionally left blank.

Prologs and Epilogs

- **Prologs are generated with the following rules**

IF	THEN
Any non-volatile FPRs are used	Save them in the FPR save area
Any non-volatile GPRs are used	Save them in the GPR save area
This routine is not a leaf procedure (i.e. it calls other routines)	Save the LR in the caller's stack frame's link area
Any of the non-volatile CR fields are used	Save the CR in the caller's stack frame's link area
A new stack frame is required (non leaf or has local variables)	Decrement R1 by the size of the required frame, save caller's R1 in the link area of the new stack frame

Table 1:

stwu R1, -X(R1)

- **Epilogs just restore all the registers saved by the prolog**
  - ***The stack frame itself is left untouched***

Notes: Prologs and Epilogs

The point of this discussion is to make you aware that each stack frame will look different, depending on what the prolog of the corresponding routine did. All that you are guaranteed is that if a prolog needs to save some registers, it will do it in a predictable way. You are never guaranteed that all registers will be saved by a prolog. Now, go back to the diagram on page 677 and trace through the prolog process, looking to see where each saved item goes.

Notice that the stack pointer is not decremented until the end of the prolog. This makes sense for the CR and LR saves, since they go into the caller's link area. But how do we get the GPRs and FPRs into the callee's stack frame?

The prolog does this by specifying negative offsets from the stack pointer for the store multiple instructions that are used to save these registers. That is why these save areas are always at the bottom of the stack frame.

Finally, note that when a stack frame is released, no changes are made to it. So, if you have a problem where routine B has returned to routine A and routine A crashed, you may still be able to look at the stack frame for B. This is why the trace subcommand shows you the top stack frame, even though the LR value in it is not current. If it is set, it shows us the last place from which the current "top" routine called another routine. That called routine has since returned to the current routine, but this bit of information may be useful in helping us find a problem.

Special Cases

- **Routines written in assembler tend to be less rigid about following linkage conventions**
  - ***If we are writing in assembler, it is usually for performance reasons***
  - ***These routines will do the bare minimum required so that they work***
    - ***Conventions can be circumvented if you know what you are doing***
    - ***Potential problem lessened because these routine are usually leaf routines***
  - ***General rule: you have to look at the prolog of a routine to be sure***

Notes: Special Cases

This page intentionally left blank.

Stack Frame Ownership

- **Which routine owns a particular stack frame?**
  - ***The routine that acquired it***
    - ***i.e. the routine that allocated the stack frame by decrementing R1***
  - ***When routine A calls routine B, B saves the LR into A's stack frame***
    - ***LR contains the address within A to which B must return***
  - ***Thus, the LR value in a stack frame points to an address within the routine that owns that stack frame***

Notes: Stack Frame Ownership

This page intentionally left blank.

The First Stack Frame

- **The first stack frame is not always useful**
  - ***Remember, the link area is used mainly by the callee***
    - ***The link register area is not in use, but it may contain a value***
    - ***Before the first call, it is uninitialized***
    - ***Otherwise, it is the address of the instruction after the last call***

Notes: The First Stack Frame

Remember that the link area of any stack frame is used mainly by routines that the frame owner calls. In particular, the link register area of the top stack frame is not currently in use. However, it may contain a value. That value is the link register value from the last call that the current routine has made up until now. So, if it is set, it should be pointing to an address within the routine that owns the stack frame.

The Top Stack Frame

- **The IAR may not be pointing within the same routine that owns the top stack frame.**
  - ***The current routine might not have acquired a stack frame.***

Notes: The Top Stack Frame

The possible reasons that the current routine may not have acquired its own stack frame:

a) It may be a leaf routine (that makes no further calls)

b) It may be an assembler routine that violates linkage conventions

c) It may not have had a chance to run its prolog before being interrupted.

Function Calls

- When calling a routine, AIX needs two pieces of information:
  - *The address of the first instruction in the routine*
  - *A "base" address to be used for symbol resolution*
- When calling a routine within the same module, calls are simple:
  - *The base address is the same*
  - *Just use bl (branch and set link register) instruction*
- Calls outside the current module are more complex
  - *Will have a different base address*

Notes: Function Calls

This page intentionally left blank.

Function Descriptors

- **Created for every routine in an executable file**
  - ***Created at compile or link time***
- **Contain three fullwords (4 bytes each):**
  - ***Pointer to actual first instruction of routine***
  - ***Pointer to Table-Of-Contents (TOC) for routine (i.e. the base address)***
  - ***Miscellaneous pointer (used by dynamically scoped languages like Pascal)***
- **Any call using a function descriptor actually issues a branch and link instruction to glink code**
  - ***Handles the usage of the function descriptor***
  - ***The compiler puts a no-op immediately after the bl***
    - ***Used later...***

Notes: Function Descriptors

This page intentionally left blank.

Table of Contents (TOC)

- **XCOFF modules have a table of contents section**
  - ***Contains the addresses of global data within the XCOFF file and any unresolved external symbols***
- **Every module has its own TOC**
  - ***A pointer to the module's TOC is put into each function descriptor created by the compiler or linker***
- **References to a particular global symbol are just offsets from the TOC**
  - ***Provides an indirect reference mechanism for all symbols***
- **TOCs are updated/relocated when bound at load time**
- **Register 2 always points to the current TOC**

Notes: Table of Contents (TOC)

This page intentionally left blank.

glink Mechanism

- **The glink code:**
  - ***Saves the caller's TOC pointer (from register 2) to the stack***
  - ***Loads the new TOC from the function descriptor***
  - ***Loads the function address from the function descriptor***
  - ***Branches to the loaded function address***
    - ***Does not set the link register***
  - ***When the called routine returns, it goes directly to the caller of glink***
- **Linker replaces the no-op after the branch and link to glink with a load of R2 to get the TOC pointer from stack**
  - ***This allows the TOC pointer to be fixed after an inter-module call***
- **Result is a way to defer the resolution of external symbols until load time.**

Notes: glink Mechanism

This page intentionally left blank.

System Calls

- **Mechanism that allows user programs to request kernel services**
  - ***Interface between user-mode programs and kernel-mode routines***
- **Forms a "protective layer" around the kernel**
  - ***System calls are the only way a user program gets into the kernel***
  - ***Allows us to write routines that need direct access to kernel data and code without giving direct access to user code***

Notes: System Calls

This page intentionally left blank.

System Call Mechanism

- **Based on the sc (system call) instruction**
  - *Generates a System Call exception that vectors to address 0xc00 (on PowerPC)*
  - *On POWER, the svc instruction is used*
    - *Vectors between 0x1000 and 0x1fff, depending on fields in opcode*
- **There is one collection of function descriptors for all system calls**
  - *They contain:*
    - *Pointer to svc_instr routine (instead of to actual routine being called)*
    - *Index into a kernel svc table (instead of a TOC pointer)*

Notes: System Call Mechanism

This page intentionally left blank.

System Call Mechanism

- **Thus, all user routines that make system calls actually call the svc_instr routine via glink**
  - ***Fixed at address 0x3600***
  - ***Contains only two instructions - one to set a CR bit indicating the type of system call, and the other, an sc instruction***

Notes: System Call Mechanism

This page intentionally left blank.

System Call Mechanism

- Normal exception handling causes the processor to switch to kernel-mode (supervisor state)
- Assumes that there is always a u_block available
  - *Holds many values that don't need to be re-saved*
    - *For example, segment registers*
- Since this is like a function call, we can use the same linkage conventions as for function calls
- Uses special save areas in uthread structure for current thread
  - *Dedicated to system call handler use*

Notes: System Call Mechanism

This page intentionally left blank.

System Call Handler

- **On entry**
  - ***Sets privileged access to the process private segment (segment 2)***
  - ***Sets privileged access to the kernel segment (segment 0)***
  - ***Sets the ut_error field in the uthread to 0***
  - ***Saves the user-mode stack pointer and Machine State Register (4.1 only)***
  - ***Switches to the kernel stack***
  - ***Finds the entry in the svc_table for the specified system call index (from register 2)***
  - ***Starts the specified kernel function (the target of the system call)***

Notes: System Call Handler

This page intentionally left blank.

System Call Handler

- **On return:**
  - ***Switches back to the user stack***
  - ***Updates the thread-specific errno variable if the ut_error field is not equal to 0***
  - ***Clears the privileged access to the kernel segment (segment 0)***
  - ***Clears the privileged access to the process private segment (segment 2)***
  - ***Performs signal processing if a signal is pending***
  - ***Reloads other segment registers from user U_adspace area***
  - ***Clears out other registers for security***
  - ***Switches back to user mode***
  - ***Returns to user program***

Notes: System Call Handler

This page intentionally left blank.

Errno

- **Understanding the Errno variable**
  - ***Used by system calls to communicate failure information to the calling program.***
  - ***thread specific errno values***

Notes: Errno

Error information returned by system calls differs from that returned by kernel services that are not system calls. System calls typically provide a return code of 0 if no error has occurred, or -1 if an error has occurred. In the later case, the error value is place in the *ut_error* field of the **uthread** structure. In some cases, when data is returned by the return code, a data value of -1 indicates error. Or alternatively, a value of NULL can indicate error, depending on the interface and function definition of the system call.

In any case, when an error condition is to be returned, the *ut_error* field should be update by the system call prior to returning from the system call function. The *ut_error* field is accessed by using the **getuerror** and **setuerror** kernel services.

Before actually calling the system call function, the system call handler sets the *ut_error* field to 0. Upon return from the system call function, the system call handler copies the value found in *ut_error* into the thread specific **errno** variable if *ut_error* was nonzero. After setting the **errno** variable, the system call handler returns to the user-mode with the return code provided by the system call function.

Kernel-mode callers of system calls must be aware of this return code convention and use the **getuerror** kernel service to obtain the error value when an error indication is returned by the system call. When system calls are nested, the system call function called by the system call handler can return the error value provided by the nested system call function or can replace this value with a new one using the **setuerror** kernel service.

AIX BOOT

Notes: AIX Boot

This page intentionally left blank.

Outline

- Types of Boot and Systems
- Boot Image and Boot Record
- Boot Mechanism
- Related Boot Facilities

Notes: Outline

This page intentionally left blank.

Objectives

- *Understand the boot process from power on until the end of the processing of the /etc/inittab file.*
- *Understand the boot process for different boot devices.*
- *Understand the basic components of a boot image.*
- *Understand how a boot image is built.*
- *Understand how the boot process differs for the three different hardware architecture types.*
- *Understand the boot related commands, utilities, and files.*

Notes: Objectives

This page intentionally left blank.

What is boot?

- **It is the process that begins when the computer is powered up and continues until the entries in the init table have been processed.**
  - ***system ROS (Read Only Storage), contains firmware that is independent of the operating system which initializes the hardware and loads AIX***
  - ***AIX begins execution after system ROS firmware finishes execution***
    - ***sets up firmware information***
    - ***kernel initialization***
    - ***RAM filesystem based configuration***
    - ***control is passed to files based in the permanent filesystem (this may be a disk or network filesystem)***
    - ***/etc/inittab entries are processed. This usually includes enabling the user login process.***

Notes: What is boot?

This page intentionally left blank.

Types of Boot

Notes: Types of Boot

This page intentionally left blank.

Various Types of boot

- devices
  - *hard disk boot*
  - *CD-ROM boot*
  - *tape boot*
  - *network boot*
- configuration
  - *standalone*
  - *diskless/dataless*
  - *operating system installation*

Notes: Various Types of Boot

This page intentionally left blank.

Hard Disk Boot

- the boot image resides on the hard disk
- the RAM filesystem contains the files necessary for configuring the hard disk(s), and then accessing the filesystems that reside in the root volume group (rootvg)
- this is the most common system configuration
- these types of systems are also known as "standalone" systems
- these types of systems may also be booted into the diagnostics functions

Notes: Hard Disk Boot

This page intentionally left blank.

CD-ROM Boot

- operating system installation
- diagnostics
- hard disk boot failure recovery/maintenance
- fast because CD-ROM is a random access media
- reliable because CD-ROM is an optical read only media
- portable and economical means for software distribution
- Most new machines ship with CD-ROM drives

Notes: CD-ROM Boot

This page intentionally left blank.

Tape Boot

- operating system installation
- hard disk boot failure recovery/maintenance
- ease of creation is convenient for customers
  - *usually used for creating bootable system backups*

Notes: Tape Boot

This page intentionally left blank.

Network Boot

- boot and install the operating system
  - *the operating system is installed on a hard disk with NIM*
  - *subsequent boots are from the hard disk*
- supported diskless/dataless configurations
- diagnostics
- hard disk boot failure recovery/maintenance
- centralized boot/filesystem servers offer convenient administration

Notes: Network Boot

This page intentionally left blank.

System Types and Boot Images

- **There are three basic hardware architecture types:**
  - ***RS6K - the "classic" IBM workstation***
  - ***RSPC - the PowerPC Reference Platform workstation***
  - ***CHRP - Common Hardware Reference Platform***
- **There are three corresponding types of boot images:**
  - ***The RS6K uses hardware ROS to build the IPL Control Block***
  - ***The RSPC and CHRP uses SOFTROS to build the IPL Control Block***
- **There are two types of Kernels loaded:**
  - ***UP (/unix->/usr/lib/boot/unix_up)***
  - ***MP (/unix->/usr/lib/boot/unix_mp)***

Notes: System Types and Boot Images

The RSPC and CHRP family of computers requires a boot image with special software known as SOFTROS, which is used to provide function that AIX requires, and is not provided by the hardware firmware. SOFTROS performs some basic hardware configuration and tests, and also sets up some data structures to provide an environment for AIX that more closely resembles the environment provided by RS6K system ROS. On CHRP systems the firmware device tree is also appended to the IPL Control Block.

The distribution CDROM that IBM provides to our customers has three boot images. There is a boot image for the RS6K computers, a second for the RSPC computers, and a third for CHRP (/ppc/chrp/bootfile.exe). This saves IBM money and is more convenient for customers. Conversely, boot/install tapes have a single boot image and are not portable between different system types (RS6K, RSPC, CHRP).

The bootinfo command can be used to determine which hardware platform a particular computer corresponds to, and whether a system supports an MP configuration.

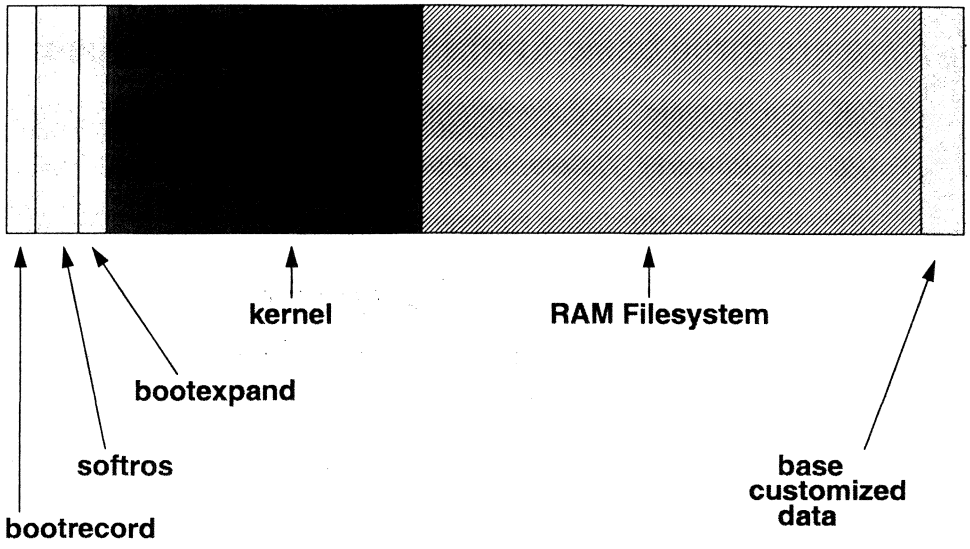
bootinfo -p will display either "RS6K", "RSPC", or "CHRP"

bootinfo -z will generate a return code of "0" that indicates the system is not MP capable, and "1" that indicates that a system is MP capable.

The RS6K, RSPC, and CHRP UP computers can use the MP Kernel, which is the method implemented for distribution media that goes to our customers. In other words, when a customer receives boot/install media from IBM, there is no need to determine whether the system is UP or MP. This boot image is created using the MP kernel. The UP kernel is more efficient for uniprocessor systems, but the strategy of a single boot image for both hardware platform types lowers distribution cost, and is more convenient for our customers.

During the AIX installation process, the bootinfo command is used to determine which kernel should be installed, and the resulting hard disk boot image is based on the appropriate kernel for the hardware platform type.

Anatomy of a compressed boot image



Notes: Anatomy of a compressed boot image

bootrecord - 512 byte block containing size and location of the boot image. The boot record is the first block on a disk or cdrom and is therefore separated from the boot image. The boot image on a disk is placed in the boot logical volume which is a reserved contiguous area.

softros - RSPC and CHRP program (name "aixmon") that performs system initialization for AIX that the hardware firmware in ROS does not provide, such as appending device information to the IPL control block.

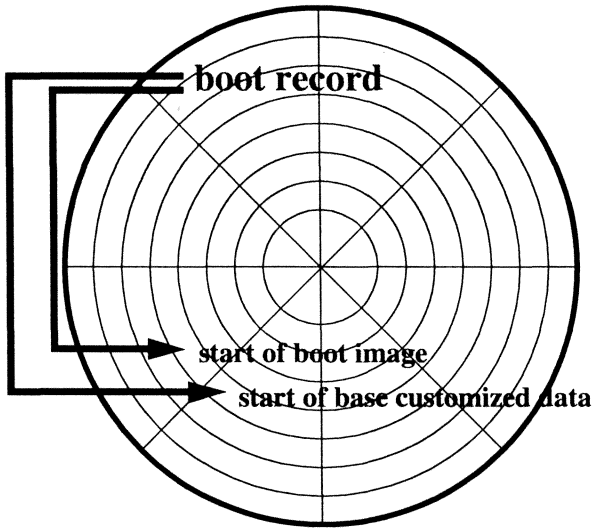
bootexpand - program to expand compressed boot image which is executed before control is passed to kernel. The compression of a boot image is optional but it is the default since the image size is less than half of an uncompressed image and requires less time to load from the media.

kernel - AIX UP or MP kernel that which control passes to after expansion by bootexpand. The kernel initializes itself and then passes control to the simple shell init in the RAM filesystem.

RAM filesystem - filesystem used during boot process, that contains programs and data for initializing devices and subsystems in order to install AIX, execute diagnostics, or to access and bringup the rest of AIX.

base customized data - area of the hard disk boot logical volume containing user configurable ODM device configuration information that is used by the system configuration process.

Disk layout of boot image



Notes: Disk layout of boot image

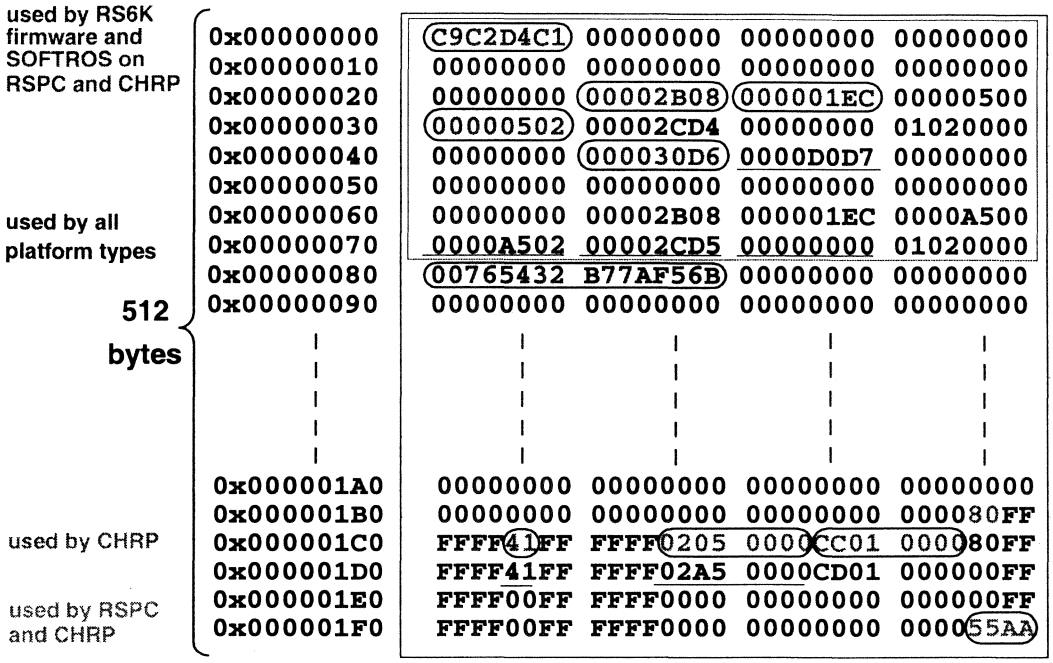
The bootrecord is the first 512 byte block on the media.

The bootrecord fields contain offsets to the remaining sections of the boot image. In the case of disk or cdrom, the remaining boot image can be located at any contiguous area. The tape and network bootimages are appended with the bootrecord.

Bootrecord offsets include starting location of softros/bootexpand/kernel, the length of the boot image, offset into kernel to start execution, and the start of the ODM savebase information.

ROS firmware uses the bootecord information to load the bootimage into memory and to pass control to the loaded image.

Example AIX Hard Disk Boot Record



Notes: Example AIX Hard Disk Boot Record

All data in the subsequent foils is in hexadecimal (or base 16) format. The arrangement on this foil is in word (32-bit) fields. The data is in the block to the right and the addresses of the left-most word of data are in the left-most column (outside the block).

These fields are defined in /usr/include/sys/bootrecord.h.

The circled values are data used during boot when the key is in the normal position. There are also corresponding values for use during boot when the key is in the service position. The values for service mode are underlined.

The data from address 0x00 to address 0x7F is signicant for RS6K systems only.

The data from address 0x1BE to address 0x1FF is significant for RSPC systems only.

The data from address 0x1CE to address 0x1FF is significant for CHRP systems only.

RS6K systems only use data from 0x00 to 0x87. RSPC and CHRP use data from the entire bootrecord. The beginning section contains the customized base data location and the end section has the information for locating the boot image.

RS6K systems only use data from 0x00 to 0x87.

Boot Record RS6K Fields

- **These fields are defined in /usr/include/sys/bootrecord.h**
  - ***field #1 (value: 0xC9C2D4C1): This field indicates whether this disk is a valid boot disk. A value of 0xC9C2D4C1 indicates validity.***
  - ***field #2 (value: 0x00002B08): This*** **boot_code_length** ***field indicates the size (number of 512 byte blocks) of the boot image, and ROS firmware uses this to determine how much data to read.***
  - ***field #3 (value: 0x000001EC): This*** **boot_code_offset** ***field is the byte offset from the start of the boot code to the first instruction.***
  - ***field #4 (value: 0x00000502): This*** **boot_prg_start** ***field is the offset (in 512 byte blocks) from the beginning of the disk to the beginning of the boot code.***
  - ***field #5 (value: 0x000030D6): This*** **basecn_length** ***field is the offset (in 512 byte blocks) from the beginning of the disk to the beginning of the saved ODM customization data.***
  - ***field #6 (value: 0x00765432_B77AF56B): This*** **pv_id** ***field is the unique LVM physical volume identifier. It is used by all platform types.***

Notes: The Boot Record Fields

The values here are represented in hexadecimal (or base 16) format. The value 0x5C means 5C interpreted as a hexadecimal number or 5 times 16 + C (or 12), which equals 92 in decimal (base 10) format.

The physical volume identifier (pvid), which is at address 0x80 is used by all three hardware platform types (RS6K, RSPC, and CHRP). This field is also used after the boot process has completed. An example of this would be device configuration and LVM configuration.

Notes: Boot Record RSPC and CHRP fields

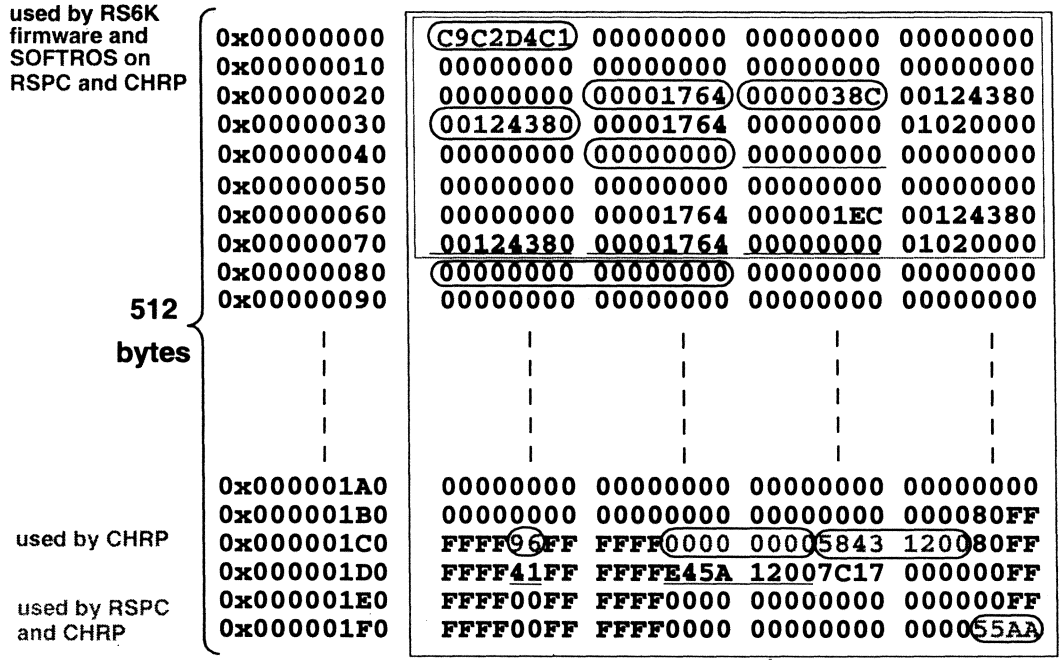
- ***field #7 (value: 0x41): This is the system indicator, and is a flag that indicates the validity of the following offset.***
- ***field #8 (value: 0x0205 0000): The relative block address, which is an hex offset (a little endian representation of the number of 512 blocks: 0x00000502) from the beginning of the disk to the beginnning of the RSPC boot code.***
- ***field #9 (value: 0xCC01 0000): Length of boot image in 512 byte blocks (little endian format).***
- ***field #10 (value: 0x55AA): A value of 0x55AA is a valid RSPC/CHRP boot signature.***

AIX V4.3.2 Kernel Internals Module 7: AIX Boot

Notes: Boot Record RSPC and CHRP fields

These fields are used only by the RSPC and CHRP firmware. On RSPC and CHRP platforms, the SOFTROS will use the RS6K fields of the boot record to complete the boot process before control is transfered to AIX.

Example AIX CD-ROM Boot Record



Notes: Example AIX CD-ROM Boot Record

The boot record fields on the AIX CD-ROM are used only for CHRP platforms.

The physical volume ID field for a CD-ROM boot record is **00000000 00000000** since the physical volume ID is only used for LVM managed disks.

The **96** value for the system indicator indicates a CD-ROM file system to CHRP firmware. This indicates to the firmware that the interpretation of the following fields is different than for hard disk boot records.

The CHRP relative block address is **0000 0000** since the boot image on CDROM is located using a filename in the CD-ROM file system, not a block location on the CD-ROM.

The **5843 1200** field is the size of the CD-ROM filesystem in little endian format, which is used by CHRP firmware.

The default CHRP boot image is in a file in the CD-ROM filesystem on the CD-ROM named **/ppc/chrp/bootfile.exe**.

The CD-ROM boot record does not contain information for the base customized ODM data, since CD-ROM boot is used only for installation and system maintenance. Base customized data is used for maintaining user customizable ODM attributes across system reboots. Since CD-ROMs are created for all systems at a particular AIX release level and not created for individual systems, there is no customizable data to put on the CD-ROM.

Boot Image Creation

- ***the bosboot command creates and connects the boot image components:***

1. set up execution environment, parse command line arguments, verify syntax and arguments
2. ensure that there is enough filesystem space
3. create a copy of the ODM database and remove all printer information from this copy
4. copy and strip the required libraries
5. use the "mkfs" command to create the RAM filesystem image, using the ODM database from step 3 and the libraries from step 4. Add other files as specified in the applicable proto files. Compress the RAM filesystem unless an uncompressed boot image is requested
6. copy and strip the kernel. Compress the kernel unless an uncompressed boot image is requested.
7. use the mkboot command to merge the kernel and the RAM filesystem image
8. if this is to be a compressed boot image, use the mkboot command to merge the decompression code and the boot image from step 7
9. copy boot image to boot device. Call the savebase command if creating hard disk boot image.
10. clean up execution environment and exit

Notes: Boot Image Creation

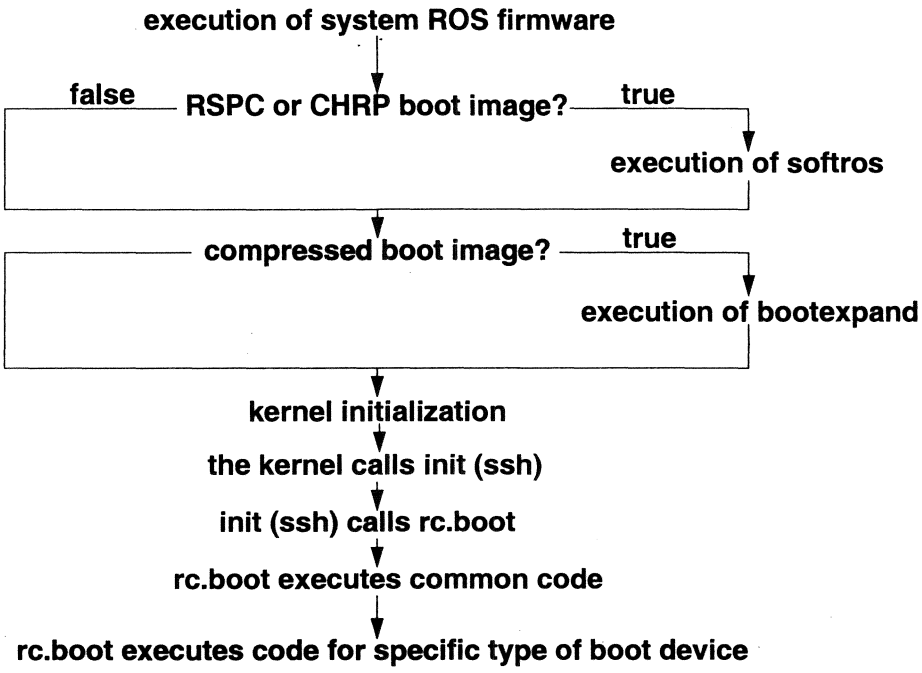
This page intentionally left blank.

Boot Mechanism

Notes: Boot Mechanism

This page intentionally left blank.

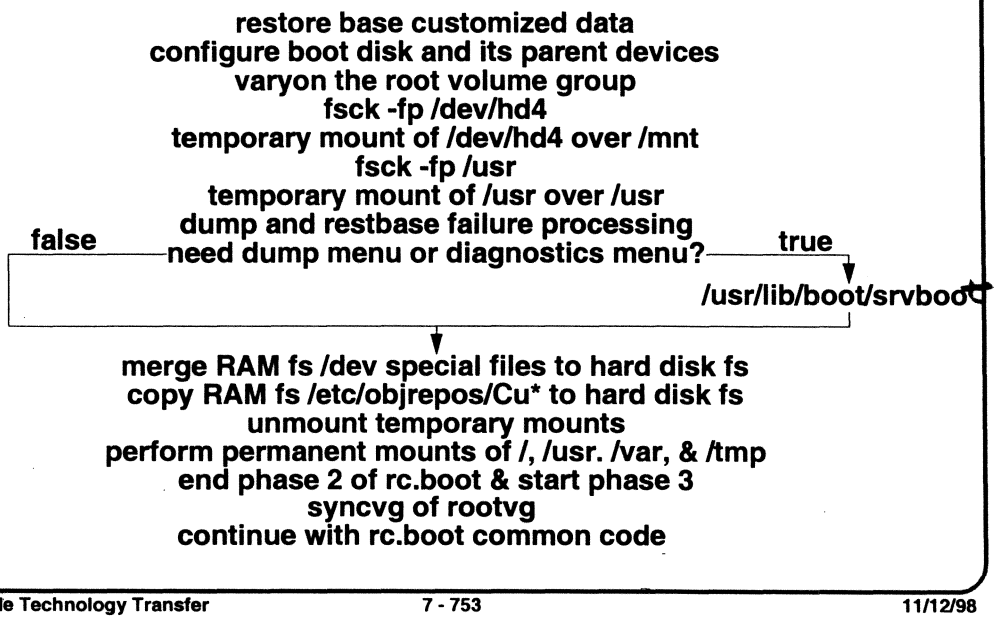
The Boot Mechanism



Notes: The Boot Mechanism

This page intentionally left blank.

Hard Disk Boot Mechanism



Notes: Hard Disk Boot Mechanism

This page intentionally left blank.

CDROM Boot Mechanism

configure CDROM drive and its parent devices
mount CDROM filesystem over /SPOT
remove most of the files from the RAM filesystem
create symbolic links from RAM fs to CDROM fs
load the streams modules
configure remaining source, target, & display devices
remove rc.boot & clean up
exec BOS installation process

Notes: CD-ROM Boot Mechanism

This page intentionally left blank.

Tape Boot Mechanism

load the streams modules

configure source, target, & display devices

set block_size attribute for all tape devices

remove files from RAM filesystem

remove rc.boot & clean up

exec BOS installation process

Notes: Tape Boot Mechanism

This page intentionally left blank.

Network Boot Mechanism

initialize environment variables with bootp reply packet information
configure network boot device and its parent devices
configure TCP/IP services
retrieve client information file via tftp
initialize additional environment variables with client.info file
NFS mount the SPOT filesystem over /SPOT/usr
copy rc.config file from server
execute rc.config file - BOS install, diskless/dataless, or diagnostics

start phase 2 of rc.boot
re-initialize environment variables with client.info file
execute rc.config file for phase 2
if diskless/dataless, end phase 2 of rc.boot
if diskless/dataless, start phase 3 of rc.boot
start portmap, rpc.statd, rpc.lockd

Notes: Network Boot Mechanism

This page intentionally left blank.

Network Boot rc.config Files

- These files reside in /usr/lib/boot/network. They are loaded from the server to the client during the network boot process. Since they are not incorporated into a boot image, this allows more flexible network boot images, and easier customization of the network boot process.
- rc.bos_inst
  - *used to configure a system for AIX installation*
- rc.dd_boot
  - *used for booting to diskless or dataless system configurations*
- rc.diag
  - *used for booting to diagnostics*

Notes: Network Boot rc.config Files

This page intentionally left blank.

Common Boot Mechanism

- these steps apply to disk boot and diskless/dataless system configurations only:

execute /usr/lib/boot/bootutil
load the streams modules
execute cfgmgr to configure all remaining devices
start dt, the graphic user interface
execute savebase
start the syncd
start the errordemon
start diagnostics if requested during phase 2
exit rc.boot

Notes: Common Boot Mechanism

This page intentionally left blank.

Related Boot Facilities

Notes: Related Boot Facilities

This page intentionally left blank.

/etc/init

- **/etc/init initializes and controls AIX processes**
  - ***The boot process, when running from the RAM filesystem, does not use the real init command. Instead, it uses /usr/lib/boot/ssh, which is renamed to /etc/init in the RAM filesystem.***
    - ***this strategy allows more efficient use of system resources during boot***
  - ***The real init is found in /usr/sbin/init. /etc/init is a symbolic link to this file. The real init begins execution during kernel newroot, which occurs at the end of phase 2 of rc.boot and before phase 3 of rc.boot. The LED display on the operator panel will display 553 during this period.***

Notes: /etc/init

This page intentionally left blank.

/etc/inittab

- **/etc/inittab is a file which the init command uses for process and system environment initialization**
  - ***the first entry initializes /etc/init***
  - ***the second entry is "rc.boot 3", which completes the system boot process***
  - ***the remaining entries start and manage various processes***
    - ***networking services***
    - ***user logins and interfaces***
    - ***daemons***

Notes: an example /etc/inittab

```
init:2:initdefault:

brc::sysinit:/sbin/rc.boot 3 >/dev/console 2>&1 # Phase 3 of system boot

powerfail::powerfail:/etc/rc.powerfail 2>&1 | alog -tboot > /dev/console # Power Failure Detection

rc:2:wait:/etc/rc 2>&1 | alog -tboot > /dev/console # Multi-User checks

fbcheck:2:wait:/usr/sbin/fbcheck 2>&1 | alog -tboot > /dev/console # run /etc/firstboot

srcmstr:2:respawn:/usr/sbin/srcmstr # System Resource Controller

rctcpip:2:wait:/etc/rc.tcpip > /dev/console 2>&1 # Start TCP/IP daemons

rcnfs:2:wait:/etc/rc.nfs > /dev/console 2>&1 # Start NFS Daemons

cron:2:respawn:/usr/sbin/cron

piobe:2:wait:/usr/lib/lpd/pio/etc/pioinit >/dev/null 2>&1  # pb cleanup

qdaemon:2:wait:/usr/bin/startsrc -sqdaemon

writesrv:2:wait:/usr/bin/startsrc -swritesrv

uprintfd:2:respawn:/usr/sbin/uprintfd

dt:2:wait:/etc/rc.dt

cons:0123456789:respawn:/usr/sbin/getty /dev/console

tty0:2:respawn:/usr/sbin/getty /dev/tty0
```

Boot Related Commands

- **alog - create and display log files**
  - *AIX versian 4 creates a boot log during each hard disk boot. The default size of this boot log is 8M, which usually accommodates the output from a single disk boot.*
  - *Display the contents of the boot log to stdout with this syntax:*

```
alog -o -t boot
```

  - *Increase the size of the boot log with this syntax:*

```
echo resizing boot log | alog -t boot -s 16384
```

- **bootinfo - display boot information (non-supported)**
  - *new flags:*
    - *"-k" to read the key switch position*
    - *"-T" to display the hardware platform type*
    - *"-z" to indicate the system's multi-processor capability*

Notes: Boot Related Commands

This page intentionally left blank.

Boot Related Commands

- **bootlist - display and/or modify boot device list**
  - ***example usage:***

```
bootlist -m service rmt cd fd0 scdisk
```

- **bosboot - build boot images**
  - ***to rebuild the disk boot image after booting from disk:***

```
bosboot -a
```

- **mkboot - low level command for building boot images**
- **savebase - saves customized data from the ODM into the boot image**

Notes: Boot Related Commands

This page intentionally left blank.

Boot Related Utilities

- */usr/sbin/bootexpand - expands a compressed boot image during boot*
- */usr/lib/boot/chramfs - dynamically expands the RAM fs during boot*
- */usr/sbin/fbcheck - configuration during the first boot after AIX installation*
- */usr/lib/boot/getrootfs - provides access to the rootvg if disk boot fails*
- */usr/lib/boot/mergedev - merges special /dev files from RAM fs to disk fs*
- */sbin/rc.boot - manages RAM filesystem boot process*
- */usr/lib/boot/restbase - restores base customized data during boot*
- */usr/lib/boot/ssh - pseudo "init" used during boot*
- */usr/lib/boot/srvboot - configures console for dump & diagnostics*
- */usr/lib/boot/update_proto - dynamically builds boot RAM fs proto files*

Notes: Boot Related Utilities

This page intentionally left blank.

Boot Related Files

- */usr/lib/boot/*.proto - specifies RAM fs contents*
  - *These files are also called "base proto files". They specify the RAM file-system files that are required in each type of device boot image. These base proto files do not include device configuration methods or device drivers.*
- */usr/lib/boot/network/*.proto - specifies network boot RAM fs contents*
  - *Similar to the above base proto files, except for network boot images.*
- */usr/lib/boot/protoext/*.proto.ext.* - lists of device support software files that are added dynamically when creating boot images*
  - *These files typically contain the device configuration files and device drivers. Here is an example from disk.proto.ext.mca.8d77.rte:*

```
hscsidd ---- 777 0 0 /usr/lib/drivers/hscsidd
cfghscsi ---- 777 0 0 /usr/lib/methods/cfghscsi
```

  - *merged into the base proto files with the* update_proto *command*

Notes: Boot Related Files

This page intentionally left blank.

Logical Volume Manager

Notes: Logical Volume Manager

This page intentionally left blank.

Outline

- LVM Components
- Physical Volumes
- Volume Groups
- Logical Volumes
- Mirroring
- Performance
- Problem Determination

Notes: Outline

This page intentionally left blank.

Contents

- LVM components
- Terms
- What does LVM give you?
- Changes between AIX 3.2 and AIX 4.1
- Performance Tuning
- Problem Determination
- Problem Recovery

Notes: Contents

This page intentionally left blank.

LVM components

- high level commands
- intermediate commands
- library programming interfaces
- LVM device driver
- disk device driver
- SCSI device driver
- SCSI adapter
- actual disk

Notes: LVM components

This page intentionally left blank.

High Level Commands

- These commands are used by SMIT and WebSM.
- These commands are used by users on the shell command line or in shell scripts.
- These commands have a high level of error checking.
- These commands may be shell scripts that call the intermediate level LVM commands.
- These commands may also be executable binaries that call the LVM library programming interfaces.

Notes: High Level Commands

These commands are those that SMIT and experienced users are meant to execute directly.

Most of the LVM commands reside in /usr/sbin. There are exceptions, such as the library commands which reside in /usr/lib/liblvm.a and /usr/lib/libsm.a and /etc/cfgvg. The high level commands are a combination of shell scripts and C binaries.

The following commands are binaries executable programs. They call the LVM library programming interfaces to do their work.

lslv, lsvg, lsvgfs, varyonvg

The following commands are shell scripts. They call the LVM intermediate level commands to do their work.

chlv, chpv, chvg, cplv, exporting, extendlv, extendvg, importvg, migratepv, mklv, mkvg, reducevg, reorgvg, rmlv, rmlvcopy, syncvg, synclvodm

The extendlv command is also is used for the mklvcopy command.

Low Level Commands

- These commands are used by the high level commands.
- These commands provide simple usage statements, giving users the impression that they are intended for end user use.
- Some level of error checking still provided, but most error checking is done by the high level commands.
- Output from these commands has no headings.

Notes: Low Level Commands

The low level commands are all C binaries. They call the LVM programming interfaces to do their work. These are the low level commands:

getlvcb, getlvname, getlvodm, getvgname, lvgenmajor, lvgenminor, lvrelmajor, lvrelminor, putlvcb, putlvodm, lchangelv, lcreatelv, ldeletelv, lextendlv, lquerylv, lreducelv, lresynclv, lchangepv, ldeletepv, linstallpv, lquerypv, lresyncpv, lcreatevg, lqueryvg, lqueryvgs, lvaryonvg, lvaryoffvg, lresynclp, lmigratepp

The commands are called by the high level command that are shell scripts. These commands unfortunately give usage statements when queried from the command line, which has given users the incorrect impression that these commands are for general use in LVM. This is not true. These commands have been "tuned" for use by the high level shell scripts. Because it is known that only high level shell scripts will be calling these intermediate commands, certain error checking is not performed by the intermediate commands. The high level commands are responsible for screening potentially dangerous errors or conditions out. The low level commands also contain some functionality that is not used and there are even some bugs that have not been fixed because the function isn't used.

Library Calls - API

- These are library calls to be used as an application program interface (API).
- Error checking is provided.
- These calls are documented in on-line documentation or hardcopy books.
- These interfaces are called by the LVM low level commands.
- These interfaces are very rarely used by application programs.

Notes: Library Calls - API

The LVM library calls are part of the LVM applications programmer interface (API) which lets programmers access the LVM layer directly through the library layer provided by /usr/lib/liblvm.a.

In our experience, very few people use this API because of the complexity of issues within the LVM. Users mostly rely on shell scripts that call the high level commands. The API library functions have names that begin with "**lvm_**" and are documented in the on-line documentation. These commands are only called in situations where the user is writing C code and does not want to access the high level shell scripts from within a C program. Or, the user applications may require a faster response from LVM than they can achieve calling the LVM shell scripts. Writing an LVM specific program using the API certainly does give a performance improvement to some applications.

Note, the LVM library API is not a thread-safe library. Thus, user programs written for threads that reference the LVM API may not work correctly.

LVM Device Driver

- This driver acts like a typical IBM device driver.
- Some error checking is provided.
- The LVM device driver is used by the journaled file system (JFS) and by the LVM library programming interfaces.
- The LVM device driver is rarely used directly by user applications.
- If the LVM device driver is used wrong, disastrous results could occur.

Notes: LVM Device Driver

The LVM device driver comes in two portions, the pinned portion and the non-pinned portion. The pinned portion is called **/usr/lib/drivers/hd_pin** and **/usr/lib/drivers/hd_pin_bot**.

The LVM device driver is either called by the jfs filesystem or the lvm library routines. When a request is received by the LVM device driver, it calls the disk device driver.

Disk Device Driver

- Access same as those for typical IBM device driver
- Sometimes other vendors use the disk device driver as pseudo driver for their products

Notes: Disk Device Driver

There are various disk device drivers in AIX. The most common disk device driver is the one for SCSI device drivers:

/usr/lib/drivers/scdisk and /usr/lib/drivers/scdiskpin

The second most common disk device driver is probably the one for the 9333 serial DASDdevice. This is found in the binaries /usr/lib/drivers/sd and /usr/lib/drivers/sdpin. In both device drivers, the portion called "pin" is that which is pinned into memory and cannot be paged out of memory. When a request from the LVM device driver is received by the disk device driver, the request is packaged and then transferred down to the SCSI device driver.

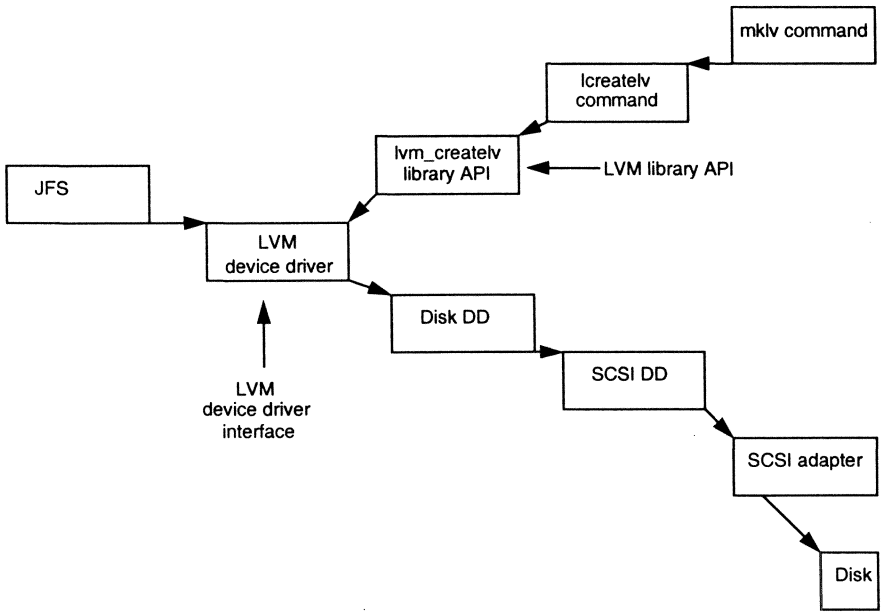
SCSI Device Driver

- Access same as those for typical IBM device driver
- Very useful to access devices directly through this interface, although people prefer not to use this access point

Notes: SCSI Device Driver

There are several SCSI device drivers for AIX and they have the common name string of "scsi" residing as part of their name. The most common of them is the original SCSI-1 device driver /usr/lib/drivers/hscsidd. The SCSI device driver takes a command from a SCSI device such as tape, disk, scanner, etc...and processes it to be sent to the SCSI device connected onto the SCSI bus. The SCSI device is device neutral, it does not know or care which device sent it a command or even what that command is. It treats all requests the same and puts them into the same type of SCSI command packaging required by all SCSI devices.

Hierarchy



Notes: Hierarchy

This page intentionally left blank.

Terms

- Physical Volumes
- Volume Groups
- Logical Volumes
- Mirrors

Notes: Terms

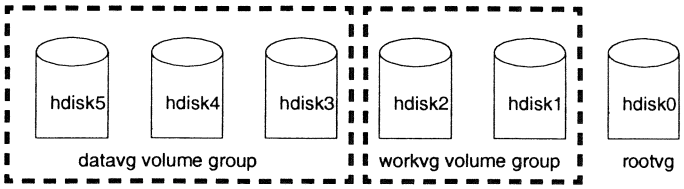
This page intentionally left blank.

Physical Volumes

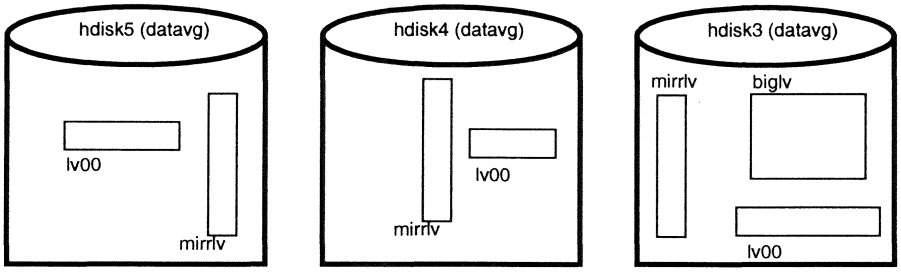
Notes: Physical Volumes

This page intentionally left blank.

Physical Volumes



On one system, many disks can make up a volume group. Disks cannot be shared between volume groups. The entire disk is dedicated to being part of one volume group.



Logical volumes reside only on volume groups. A logical volume can span multiple disks within a volume group (but cannot span multiple volume groups): lv00. Or, a logical volume can just reside on one disk: biglv. Finally, a logical volume can have multiple copies of itself: mirrlv.

Notes: Physical Volumes

Disk Type Independence

- LVM treats all types of physical disks similarly.
- Disks names do not reflect the type of physical disk.
- Removable media have special LVM treatment.

Notes: Disk Type Independence

The basic design of the interaction between LVM and the disk device driver has always assured that LVM's use of the disk would behave the same regardless of the type of disk being used in LVM. Thus, a disk drive such as the serial DASD drive (9333 drives) behaves the same in LVM as the standard SCSI drive, although they use different disk and adapter device drivers. LVM just concentrates on the special file that represents the disk drive. Although it is not a requirement by LVM, almost all the disks are configured with the name hdiskX. The only exception to this transparent use of disks by LVM are the read/write optical disk drives. There is special code in the high-level LVM commands which checks for the existence of optical drives. Because of the removability (and portability) of optical media, special handling must take effect to guarantee that LVM knows the true identity of the optical disk drives.

Creation of Physical Volume IDs

- PVIDs are created by LVM using a system specific ID and the date and time of PVID creation.
- It is extremely unlikely that two physical volumes have the same PVID, even on different systems.
- This is important because LVM supports moving of volume groups and their physical volumes between machines.
- Copying disk contents (including PVIDs) between disks is very dangerous.

Notes: Physical Volumes

PVID's and how they configure

When a disk is configured to a system for the first time, it shouldn't have a PVID if it is a brand new disk. When it is used in a volume group, the user sees a message to the effect of:

Making this disk a physical volume

Which is another way of saying that a PVID is being placed onto the disk. The PVID is an amalgamation of the machine's serial number (from the systems EPROMs) and the date that the PVID is being generated. This combination tries to achieve an extremely low chance of two disks being created with the same PVID. Finally, when a system is booted, the disk configurator goes and looks at the PVID sitting on each disk platter and then compares that to an entry in ODM. If the entry is found, then the disk is given the **hdisk** name that is associated with the ODM entry for the PVID. If there is no PVID, the configuration routines will automatically assign the next free **hdisk** name from the pool of "free" hdisk names. Note that if a hdisk has been removed with the "rmdev -l hdiskX -d" command, then this hdisk name will be available for reuse by a later disk.

Warning: Some users use the "dd" command to copy the entire contents of one disk to another disk. This is an unsupported method of "quick install". What the user forgets in this action is that they are literally copying over the PVID of the first disk onto the platter of the second disk. The extremely rare consequence of this "dd" is that the user may have, in the distant future, two disks attached to the same RS/6000 with the same PVID. When this occurs, EXTREME confusion will occur in the LVM.

Physical Volume ID Determination

- Physical Volume IDs (PVIDs) are stored in ODM and on the hard disk.
- The LVM commands usually use the PVID in the ODM.
- Sometimes the PVID in the ODM can be different than the PVID on the hard disk.
- Manual intervention is required to make the PVIDs consistent.

Notes: Physical Volume ID Determination

Sometimes it can get confusing as to the true name of a disk and how it is recognized by the system. This usually occurs during service calls or after a system has sustained a crash. The ODM is used to attach an hdiskX name to a physical volume ID (PVID) of a disk. The PVID is the "soft serial number" of a disk that is created usually just once during the lifetime of a disk's usage on AIX. The only time this PVID needs to be recreated is when someone accidently overwrites it with an inadvertent "dd" command or they purposely scrub the disk (PVID and all) with a low level diagnostics reformat command. Depending on the situation, sometimes the ODM can be confused as to the true identity of a disk. There are two methods to see the true PVID that resides on a disk. You can use the lquerypv command:

/usr/sbin/lquerypv -h /dev/hdiskX

Which shows the hex dump of the first block of the disk platter. On row 0x80, the PVID is shown, usually with the first two columns containing the hexadecimal PVID, and the last two columns consisting of all zeros.

Number of Physical Partions per Disk

- Mirroring requires LVM to maintain status for potentially every physical partition on a disk.
- LVM thus limits the number of physical partitions that can be represented on all physical volumes of a volume group.
- Normal volume groups can represent status for 32,512 physical partitions.
- Big volume groups can represent status for 130,048 physical partitions.
- At volume group creation time the maximum number of disks is specified, which determines how many physical partitions are available for each disk.

Notes: Number of Physical Partitions per Disk

In the design of LVM, each logical partition maps to one physical partition. The design of LVM limits the number of physical partitions that LVM can track per disk.

By default, a volume group is created with 4 megabyte physical partitions and a maximum of 32 physical disks, and the volume group is created as a normal volume group. The default choices would yield a volume group that can only support disks up to about 4 gigabytes in size (1016 maximum physical partions times the 4 megabyte physical partition size).

The physical partion size of all disks in a volume group can be set to one of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024 megabytes using the "-s" option to "mkvg".

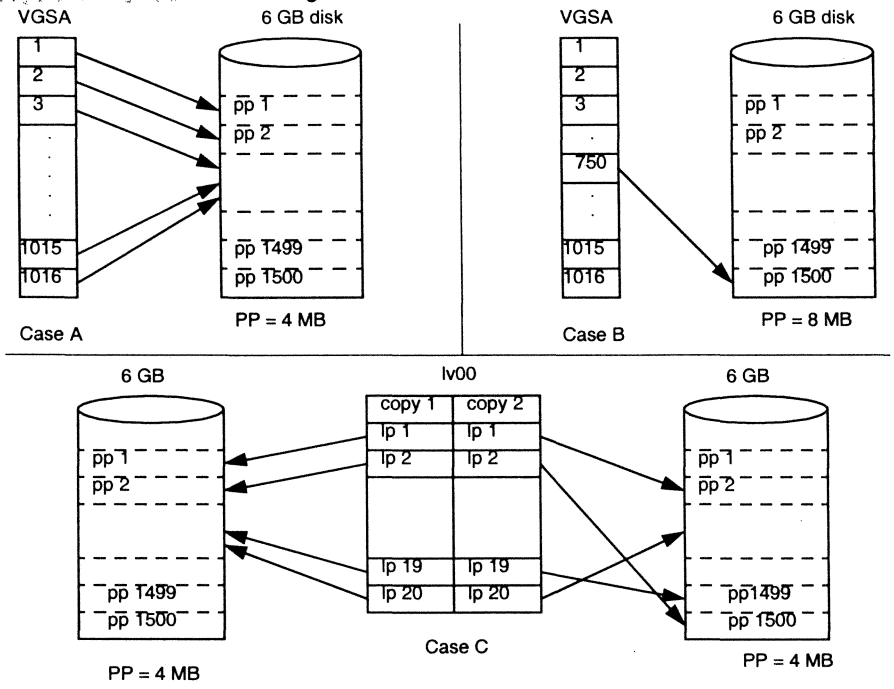
If a disk is added to a volume group that would require more than the maximum number of physical partitions, the disk addition will fail with a warning message that the physical partition size needs to be increased. There are two instances where this limitation will be enforced. The first case is when the user tries to use "mkvg" to create a volume group where the number of physical partitions on one of the disks in the volume group would exceed the maximum. The second case is where the disk which violates the physical partition limitation is attempting to join a pre-existing volume group with the "extendvg" command. The user can either recreate the volume group with different parameters create another volume group for the new disks.

On AIX 4.3.1, the maximum number of physical disks in a volume group can be reduced to increase the number of physical partitions available to each disk. The default number of disks in a normal volume group is 32 and for a big volume group is 128. In both cases, this yields a maximum number of physical partitions per physical volume of 1016.

On AIX 4.3.2, the user can also create a "big" volume group, which increases the number of physical partitions available by a factor of four.

The operating system installation code has to create the root volume group, "**rootvg**", without customer interaction. By default, it creates a normal volume group with 32 maximum disks. If it detects that the **rootvg** drive is larger than 4 gigabytes, it will change the physical partition size until the entire disk capacity can be mapped to the available 1016 tracks. This install change also implies that all other disks added to **rootvg**, regardless of size, will also be defined at that new physical partition size.

Physical Volumes



Notes: Physical Volumes

These three cases show the problems and remedies of the maximum number of physical partition problem. In all cases the default volume group parameters are used (maximum 32 disks, 4 megabyte physical partitions, and a "normal" volume group).

Case A shows what would happen if the physical partition limitation were not enforced. In the 6 gigabyte disk, the status for all physical partitions starting with partition 1017 would no longer be tracked by the LVM. This is rather important because it implies that at least one-third of the disk would not be correctly monitored by LVM mirroring logic.

Case B shows the corrected case and how, by changing the physical partition size to 8 megabytes one can have more than enough LVM information to monitor all partitions in the 6 gigabyte disk.

Case C shows the real reason that the physical partition limitation is important. The user cannot assume that the first 1016 partitions of a disk are going to be used for a logical volume. The user cannot say, "I won't worry about the physical partition limitation because I'll only mirror using the first 4 gigabytes of the disk". Unless the user takes time to create logical volumes with map files, there is no way to guarantee which physical partitions will be used for which logical partition with which mirror copy. In Case C, copy 1 shows the idealistic mapping and copy 2 shows a "worst case" mapping.

Volume Groups

Notes: Volume Groups

This page intentionally left blank.

Volume Groups

- Portability of volume groups
- Mix and Match disks
- Volume Group Descriptor Area
- Volume Group Status Area
- Quorum
- VGID
- A Graphical View of VGDA expansion

Notes: Volume Groups

This page intentionally left blank.

Portability of Volume Groups

- A volume group can be "exported" on one system and "imported" on another system.
- This allows for data migration between systems and data preservation across system reinstall.
- Volume group and logical volume name collisions are resolved by LVM by changing names on the imported volume group.

Notes: Volume Groups

One useful attribute of LVM is the ability of the user to take a disk or sets of disks that make up a volume group and take it to another RS/6000 system and introduce the information created on another machine onto the second machine. This ability is provided through the Volume Group Descriptor Area (VGDA) and the logical volume control block (LVCB). The design of LVM also allows for accidental duplication of volume group and logical volume names. If on the new machine, the volume group or logical volume names being imported already exist, then LVM will generate a distinct volume group or logical volume name.

Volume Groups

- Different types of disks can be combined into a single volume group.
- RAID systems, SCSI disks, and SSA disks can be supported in this way.

Notes: Volume Groups

As mentioned before, LVM allows the user to attach disks to volume group, regardless of what type of physical device it true is or what type of device driver is used to operate the device. Thus, RAID systems, serial DASD drives, and the plain SCSI drives can all make up one volume group that may reside across several adapters. The physical location of each drive and the true nature of the drive doesn't matter to LVM as long as the disk device drivers follow a certain format required by LVM in order to create logical volumes on those drives.

Volume Group Descriptor Area

- This is also known as the "VGDA".
- The VGDA resides on the front of all disks in the volume group.
- The VGDA contains information on the logical volumes in the volume group.
- The VGDA on every disk in the volume group contains the exact same data.
- Thus, the VGDA on one disk contains all the information about the volume group and all logical volumes on every disk in the volume group.

Notes: Volume Group Descriptor Area

The VGDA is an area at the front of each disk which contains information about the volume group, the logical volumes that reside on the volume group and disks that make up the volume group. For each disk in a volume group, there exists a VGDA concerning that volume group. This VGDA area is also used in quorum voting. The VGDA contains information about what other disks make up the volume group. This information is what allows the user to just specify one of the disks in the volume group when they are using the "importvg" command to import a volume group into an AIX system. The importvg will go to that disk, read the VGDA and find out what other disks (by PVID) make up the volume group and automatically import those disks (and their ODM information) into the system. The information about neighboring disks can sometimes be useful in data recovery. For any logical volume that exists on a disk, the VGDA gives information about that logical volume so that any time some change is made to the status of the logical volume (creation, extension, or deletion), then the VGDA on that disk and the others in the volume group must be updated.

In some instances, the user will experience a problem adding a new disk to an existing volume group or in the creation of a new volume group. The warning message provided by LVM will be: "Not enough descriptor space left in this volume group. Either try adding a smaller PV or use another volume group". When the user creates a volume group, the "mkvg" command defaults to allowing the new volume group to have a maximum of 32 disks (128 for big volume groups) in a volume group. However, as bigger disks have become more prevalent, this limit is usually not achieved because the space in the VGDA is used up faster, as it accounts for the capacity of the bigger disks. This maximum VGDA space is a fixed size which is allocated when a volume group is created. Large disks require more management mapping space in the VGDA, which causes the number and size of available disks to be added to the existing volume group to shrink. When a disk is added to a volume group, not only does the new disk get a copy of the updated VGDA, but as mentioned before, all existing drives in the volume group must be able to accept the new, updated VGDA.

The exception to this description of the maximum VGDA is **rootvg**. In order to provide AIX users more free space, when **rootvg** is created, "mkvg" does not use the maximum limit of 32 disks that are allowed into a volume group. Instead, the number of disks picked in the install menu of AIX is used to determine the maximum number of disks for **rootvg**. This number is 7 for one disk and one more for each additional disk picked. (i.e. If you pick two disks, the number is 8. If you pick three disks, the number is 9, etc.) This limit does not mean the user cannot add more disks to **rootvg** in the post-install phase. The amount of free space left in a VGDA, and thus the number and size of the disks added to a volume group, depends on the size and number of disks already defined for a volume group. However, this smaller size during **rootvg** creation implies the user will be able to add fewer disks to **rootvg** than compared to a non-**rootvg** volume group.

Volume Group Status Area

- The Volume Group Status Area (VGSA) records information on stale partitions for mirroring.
- The VGSA represents the stale partition information for only the disk containing the VGSA.
- The VGSA is a bit array with one bit for every physical partition on a disk.

Notes: Volume Group Status Area

The Volume Group Status Area (VGSA) is comprised of 127 bytes, where each bit represents one of the physical partitions that reside on each disk. The bits of the VGSA are used as a quick bit-mask to determine which physical partitions, if any, have become stale. This is only important in the case of mirroring where there exists more than one copy of the physical partition. Stale partitions are flagged by the VGSA. Unlike the VGDA, the VGSA's are specific only to the drives which they exist. They do not contain information about the status of partitions on other drives in the same volume group. The VGSA is also used to determine which physical partitions must undergo data resyncing when mirror copy resolution is performed.

Quorum Policy

- The quorum policy is a policy that if VGDAs differ between physical disks, the majority of VGDAs are taken to have the correct information.
- More than half of the VGDAs in the volume group must be available for the volume group to be used.
- Each disk has one "vote" because it has one VGDA.
- The two disk case is a special case where one disk is given two VGDAs and thus two "votes".
- The two disk case allows the volume group to be used if one of the disks (the one with two votes) is available.
- Quorum policy can be ignored with manual intervention.

Notes: Quorum Policy

Quorum is a sort of "sanity" check that LVM uses to resolve possible data conflicts and prevent data corruption. Quorum is a method by which 51% or more quorum votes must be available to a volume group before LVM actions can continue. Quorum is issued to a disk in a volume group according to how the disk was created within the volume group. When a volume group consists of one disk, there are two VGDA's on that disk. Thus, this single disk volume group has a quorum vote of 2. When another disk is added to the volume group with an "extendvg", then this new disk gets one VGDA, but the original, first disk still retains the two VGDA's. When the volume group has been extended to three disks, the third disk gets the spare VGDA sitting on the first disk and then each disk has a quorum vote of 1. Every disk after the third disk is automatically given one VGDA, and thus one vote.

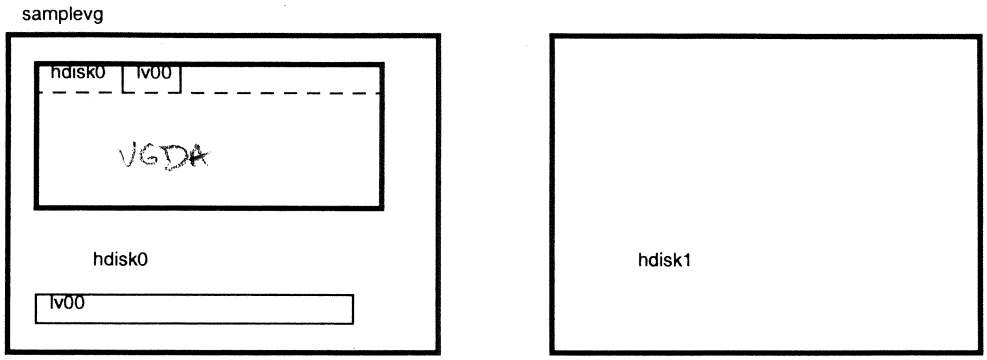
Volume Group ID (VGID)

- The VGID is a "soft serial number" for the volume group.
- The VGID is used in the LVM to identify the volume group.
- Logical volume IDs are created based upon the VGID.

Notes: Volume Group ID (VGID)

Just as the PVID is a soft serial number for a disk, the VGID is the soft serial number for the volume group. It is this serial number, not the volume group's ASCII name, which all low level LVM commands reference. Additionally, it is the basis for the LVIDs created on that volume group.

Volume Group Expansion



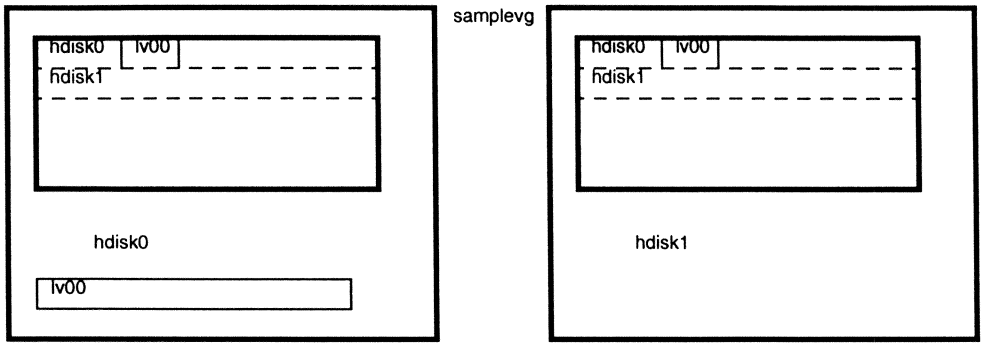
before integration of hdisk1 into the volume group

Notes: Volume Group Expansion

This shows how the VGDA grows from one disk to another and how information is carried in the VGDA which cross-references information among the disks in a volume group. The exact VGDA structure is not defined in this section because it is quite complex and contains proprietary information.

In these examples, you see how the VGDA is used to monitor disks and logical volumes that belong to a volume group.The volume group, samplevg, contains only hdisk0. The example also shows how the VGDA holds the map for the logical volume lv00. There is no VGDA represented on hdisk1 since it is not part of the samplevg. However, it may be the case that hdisk1 contains some long-forgotten VGDA from a previous volume group. But since the VGDA for samplevg has no record of hdisk1 belonging to the volume group, hdisk1 is never examined for a VGDA for samplevg.

Volume Group Expansion

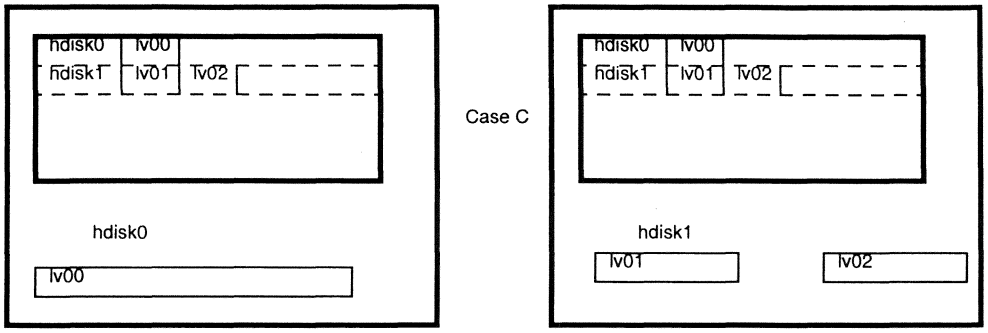


after integration of hdisk1 into the volume group

Notes: Volume Group Expansion

This is an example of the aftermath of an extendvg of samplevg into hdisk1. A copy of the VGDA from hdisk0 is placed onto hdisk1. But, notice that the VGDA has also been updated on hdisk0 to reflect the inclusion of a new disk into the volume group. Although lv00 does not exist on hdisk1, the VGDA on hdisk1 will have a record of the lv00 mapping sitting on another disk since it's part of the same volume group.

Adding Logical Volumes



after creation of new logical volumes on hdisk1

Notes: Adding Logical Volumes

This shows the result of the creation of two logical volumes, lv01 and lv02, on hdisk1. This is to show how the VGDA sitting on hdisk0 is updated with the activity that is being conducted on another disk in the volume group.

Logical Volumes

Notes: Logical Volumes

This page intentionally left blank.

Logical Volume Membership

- A logical volume must be a member of a volume group.
- A logical volume can only be a member of a specific volume group.
- All mirrors of a logical volume must be in the same volume group.

Notes: Logical Volume Membership

Logical volumes that reside on a volume group and only exist in that volume group. Logical volumes are not allowed to be mirrored to other volume groups and cannot be spanned from one volume group to another volume group. This is in part to the information held on a logical volume by the VGDA. The VGDA can't be expected to not only track a logical volume in its own volume group and then additionally track the logical volume (and more important its status) on another volume group.

Logical Volume Partition Placement

- **The user can specify where a logical volume's logical partions get located.**
  - inter-policy - ***controls on which disks the logical volume's logical partions can be placed***
  - intra-policy - ***controls in which region of a disk the logical volume's logical partions can be placed***
  - map file - ***controls which specific physical partion will be used for each logical partition in the logical volume***

Notes: Logical Volume Partition Placement

The creation of the logical volume allows many levels of control, from no control to disk specific control to partition location control. The main binary which determines how the partitions are allocated to the user's request is /usr/sbin/allocp. The binary takes into account user specific requirements at the command line and the default values, such as the intra-policy and the inter-policy. With no specific arguments, such as the disk(s) where the logical volumes "should" be created on, allocp will just use the available defaults as the blueprint to determine how the logical volume is allocated. If the user gives a list of disks where they would like the logical volume to be allocated, allocp will create the logical volume with those disks as its parameters. If it cannot fulfill these disk requirements, then the "mklv" command will fail. Finally, with map files, the user can tell the logical volume which exact physical partition on the disk they wish the logical volume to be created. Implicit in this fact is the ability to control the ORDER which these physical partitions are allocated. People with their own theories of optimal data access tend to try to control the logical volume formation at the physical partition level.

Logical Volume Partition Placement

- The user can view where a logical volume's logical partitions are located.
- The user can view which disks are used for a logical volume's logical partitions are located.
- The user can view which physical partitions on a disk are used for a logical volume's logical partitions.

Notes: Logical Volume Partition Placement

The LVM commands provide many ways of letting the user view the same logical volume. They can usually just be simply deduced, after some experience in using LVM. For instance, if I want to look at how a logical volume is laid out onto one disk, I simply type: "lspv -M hdiskA | grep lvX". Or, I can type "lslv -p lvX". Or, I can use "lslv -m lvX". Or, I can use "lsvg -l vgname | grep lvX". The point is that there is more than one way the logical volume's location can be viewed by the many LVM high level commands.

Non-Typical Logical Volumes

- **Most logical volumes are used for JFS file systems.**
- **There are other types of logical volumes:**
  - log - ***used for JFS logging***
  - dump - ***used for kernel crash dumps***
  - paging - ***used for paging spaces***
  - boot - ***used for system boot images***

Notes: Non-Typical Logical Volumes

There are a few logical volumes that exist on the system that are used and accessed in the typical manner. Besides the log logical volume (used by jfs), there are the dump device logical volume, the boot logical volume, and the paging device logical volume. The paging device is often used as a dump device, but a separate dump logical volume can be created. Using the paging device as a dump device can "lighten" the system's resource usage. Not only do you free up disk space formally used by the dump logical volume, but you've also now guaranteed that the dump device is large enough to capture the dump image if you do have a system failure. However, there is a side-effect to this change. When the user tries moving their original paging device (hd6), they cannot do so even though they have correctly turned off their paging device and rebooted the machine. The reason is that the dump device has a lock on the logical volume that prevents the removal of the dormant paging device. This is easily fixed by resetting the dump device with the "sysdumpdev" command.

Logical Volume ID (LVID)

- This is the "soft serial number" for the logical volume.
- The LVID is created from the VGID of the volume group that it belongs to and a number indicating the order of its creation within the volume group.

Notes: Logical Volume ID (LVID)

The LVID is the soft serial number used to represent the logical volume to the LVM libraries and low level commands. The LVID is created from the VGID of the volume group, a decimal point, and a number which represents the order which the logical volume was created on the volume group.

Logical Volume Control Block (LVCB)

- The LVCB occupies first 512 bytes of the logical volume.
- The LVCB can be damaged by direct modification of the logical volume.
- The LVCB is not required for most LVM operations.
- Most LVM operations use the ODM for information that is duplicated in the LVCB.

Notes: Logical Volume Control Block (LVCB)

The logical volume control block (LVCB) consists of the first 512 bytes of a logical volume. This area holds important information such as the creation date of the logical volume, information about mirrored copies, and possible mount points in a journaled filesystem. Certain LVM commands are required to update the LVCB, as part of completeness algorithms in LVM. The old LVCB area is first read and analyzed to see if it is a valid LVCB. If the information is verified as valid LVCB information, then the LVCB is updated. If the information is not completely valid, then the LVCB update is not performed and the user is given the warning message:

Warning, cannot write lv control block data

Most of the time, this is a result of database programs accessing the raw logical volumes (and thus bypassing the journaled filesystem) as storage media. When this occurs, the information for the database is literally written over the LVCB. Although this may seem fatal, it is not the case. Once the LVCB has been overwritten, the user can still extend a logical volume, create mirrored copies of a logical volume, remove the logical volume, and create a journaled filesystem with which to mount the logical volume (note that this will destroy any data sitting in the LVCB area).

However, there is a limitation caused by this deletion of the LVCB. The logical volumes with deleted LVCB's face possible, incomplete importation into other AIX systems. During an "importvg", the LVM command will scan the LVCB's of all defined logical volumes in the volume group to be imported. Surprisingly, if the LVCB is deleted, the imported volume group will still define the logical volume to the new AIX system which is accessing this volume group, and the user can still access the raw logical volume. However, any journaled filesystem information is lost and the logical volume and its associated mount point won't be imported into the new AIX system. The user must create new mount points and the availability of previous data stored in the filesystem is NOT assured. Additionally, all copies of the logical volume (mirrors), except the original copy, are put into an unreliable state. The mirrors are physically there, but some of the lvm commands which depend on their information from the LVCB will not realize that the mirrors still exist. Finally, with an erased LVCB, the output from the "lslv" command might be misleading or unreliable.

Mirroring

Notes: Mirroring

This page intentionally left blank.

Mirror Copies

- All copies of a mirrored logical volume have identical data maintained.
- All copies of a mirrored logical volume are considered equivalent.
- One exception is the primary copy of a sequentially mirrored logical volume.

Notes: Mirror Copies

When discussing mirrors in LVM, it is easier to refer to each copy, regardless of when it was created, as a copy. the exception to this is when one discusses sequential mirroring. In sequential mirroring, there is a distinct primary copy and secondary copies. However, the majority of mirrors created on AIX systems are of the parallel type. In parallel mode, there are no primary or secondary mirrors. All copies in a mirrored set are just referred to as copies, regardless of which one was created first. Since the user can remove any copy from any disk, at any time, there can be no ordering of copies.

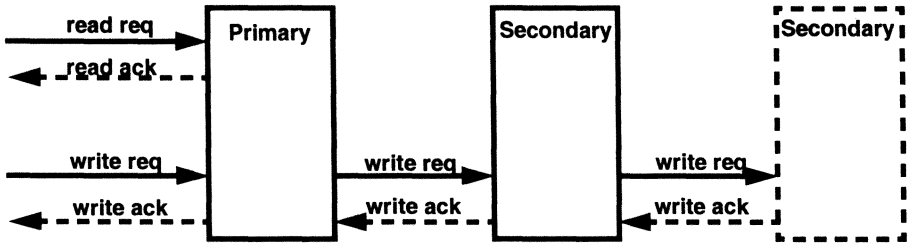
Mirror Copy Partition Staleness

- A "staleness" attribute is kept for every partition of every mirror copy of a mirrored logical volume.
- If a write to a mirror copy fails, the partition containing the data is marked stale.
- No writes are every attempted to a partition that is marked stale.
- *syncvg* causes mirror copy partitions to be made identical to non-stale copies.
- When a stale partition is made to have the same data as all non-stale copies, the partition is marked not stale again.

Notes: Mirror Copy Partition Staleness

The idea of a mirror is to provide an alternate, physical copy of information. If one of the copies has become unavailable, usually due to disk failure, then we refer to that copy of the mirror as going "stale". Staleness is determined by the LVM device driver when a request to the disk device driver returns with a certain type of error. When this occurs, the LVM device driver notifies the VGSA of a disk that a particular physical partition on that disk is stale. This information will prevent further reads or writes from being issued to physical partitions defined as stale by the VGSA of that disk. Additionally, when the disk once again becomes available (suppose it had been turned off accidentally), the synchronization code knows which exact physical partitions must be updated, instead of defaulting to the update of the entire disk. Certain high level commands will display the physical partitions and their stale condition so that the user can realize which disks may be experiencing a physical failure.

Sequential Mirroring

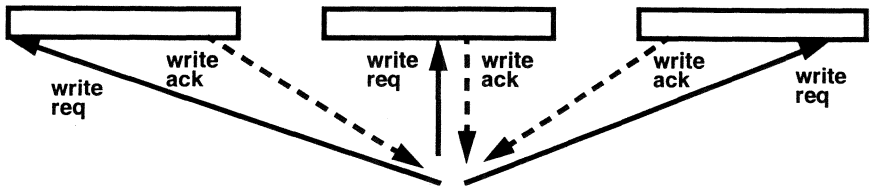


- Reads always come from the primary mirror copy.
- Writes go first to the primary mirror copy, then to the secondary mirror copies in sequential order.
- The acknowledgment of a write occurs after all complete.
- Sequential mirroring ensures that the most up to date copy of the data is always returned on a read.

Notes: Sequential Mirroring

Sequential mirroring is based on the concept of an order within mirrors. All read and write requests first go through a PRIMARY copy which services the request. If the request is a write, then the write request is propagated sequentially to the SECONDARY drives. Once the secondary drives have serviced the same write request, then the LVM device driver will consider the write request complete.

Parallel Mirroring



- Reads are sent only to the disk with the fewest requests.
- Writes go to every mirror copy at the same time.
- The write is complete after all disks acknowledge.
- Parallel mirroring completes write operations in the shortest amount of time.
- Parallel mirroring attempts to complete reads in the shortest amount of time and balance disk load.

Notes: Parallel Mirroring

In parallel mirroring, all copies are of equal ordering. Thus, when a read request arrives to the LVM, there is no first or favorite copy that is accessed for the read. A search is done on the request queues for the drives which contain the mirror physical partition that is required. The drive that has the fewest requests is picked as the disk drive which will service the read request. On write requests, the LVM driver will broadcast to all drives which have a copy of the physical partition that needs updating. Only when all write requests return will the write be considered complete and the write-complete message will be returned to the calling program.

It becomes obvious that there is a definite performance advantage to use parallel mirroring over sequential mirroring. The question arises why anyone would be interested in using Sequential Mirroring. The answer is that since there is a definite and defined Primary copy, this will be the first disk always read during a reboot. Thus, it will be the "source" mirror copy used when mirror resync occurs. This implication will become clear in the section on Mirror Write Consistency Check.

LVM Flexibility

- Volume groups and logical volumes can be expanded and deleted at any time.
- Data integrity mechanisms and be user customized.
- Logical volumes' capabilities can be exploited under JFS filesystems.
- Logical volumes' capabilities can be exploited by applications that expect to use raw disks.
- Logical volumes' allocation can be user customized.

Notes: LVM Flexibility

Typical UNIX operating systems have static filesystems that require the archiving, deletion, and recreation of larger filesystems in order for an existing filesystem to expand. LVM allows the user to add disks to the system without bringing the system down and allows the real-time expansion of the filesystem through the use of the logical volume. All filesystems exist on top of logical volumes. However, logical volumes can exist without the presence of a filesystem. When a filesystem is created, the system first creates a logical volume, then places the journaled filesystem (jfs) "layer" on top of that logical volume. When a filesystem is expanded, the logical volume associated with that filesystem is first "grown", then the jfs is "stretched" to match the grown filesystem.

The user has the ability to control which levels of data integrity checks are placed in the LVM code in order to tune the system performance. The user can change the mirror write consistency check, create mirroring, and change the requirement for quorum in a volume group.

The logical volume is a logical to physical entity which allows the mapping of data. The jfs maps files defined in its filesystem in its own logical way and then translates file actions to a logical request. This logical request is sent to the LVM device driver which converts this logical request into a physical request. When the LVM device driver sends this physical request to the disk device driver, it is further translated into another physical mapping. At this level, LVM does not care about where the data is truly located on the disk platter. But with this logical to physical abstraction, LVM provides for the easy expansion of a filesystem, ease in mirroring data for a filesystem, and the performance improvement of file access in certain LVM configurations.

As stated before, the logical volume can run without the existence of the jfs filesystem to hold data. Typically, database programs use the "raw" logical volume as a data "device" or "disk". They use the LVM logical volumes (rather than the raw disk itself) because LVM allows them to control which disks the data resides, allows the flexibility to add disks and "grow" the logical volume, and gives data integrity with the mirroring of the data via the logical volume mirroring capability.

The user can create logical volumes, using a map file, that will allow them to specify the exact disk(s) the logical volume will inhabit and the exact order on the disk(s) that the logical volume will be created in. This ability allows the user to tune the creation of their logical volumes for performance cases.

Data Integrity

- Mirroring can be used to handle disk I/O problems.
- Mirror write consistency check (MWCC) can be used to ensure all mirrors have consistent data.
- Stale mirror copies will be detected and corrected when *syncvg* is run or when *varyonvg* is run on the volume group.
- Write verify can be used to ensure that written data is actually readable from the disk.

Notes: Data Integrity

AIX allows up to three copies of a logical volume and the copies may be in sequential or parallel arrangements. Mirrors improve the data integrity of a system by providing more than one source of identical data. With multiple copies of a logical volume, if one copy cannot provide the data, one or two secondary copies may be accessed to provided the desired data.

Mirror Write Consistency Check (MWCC) is a method of tracking the last 62 writes to a mirrored logical volume. If the AIX system crashes, upon reboot the last 62 writes to mirrors are examined and one of the mirrors is used as a "source" to synchronize the mirrors (based on the last 62 disk locations that were written). This "source" is of importance to parallel mirrored systems. In sequentially mirrored systems, the "source" is always picked to be the primary disk. If that disk fails to respond, the next disk in the sequential ordering will be picked as the "source" copy. There is a chance that the mirror picked as "source" to correct the other mirrors was not the one that received the latest write before the system crashed. Thus, the write that may have completed on one copy and incomplete on another mirror would be lost.

AIX does not guarantee that the absolute, latest write request completed before a crash will be there after the system reboots. But, AIX will guarantee that the parallel mirrors will be consistent with each other. Data consistency is guaranteed with MWCC, but the getting the latest possible write can be assured by using the primary mirror for reads in sequential mirroring.

The Volume Group Status Area (VGSA) tracks the status of 1016 physical partitions per disk per volume group. During a read or write, if the LVM device driver detects that there was a failure in fulfilling a request, the VGSA will note the physical partition(s) that failed and mark that partition(s) "stale". When a partition is marked stale, this is logged by AIX error logging and the LVM device driver will know not to send further partition data requests to that stale partition. This saves wasted time in sending i/o requests to a partition that most likely will not respond. And when this physical problem is corrected, the VGSA will tell the mirror synchronization code which partitions need to be updated to have the mirrors contain the same data.

There is a capability in LVM to specify that you wish an extra level of data integrity is assured every time you write data to the disk. This is the ability known as write verify. This capability is given to each logical volume in a volume group. When you have write verify enabled, every write to a physical portion of a disk that's part of a logical volume causes the disk device driver to issue the Write and Verify scsi command to the disk. This means that after each write, the disk will reread the data and do an IOCC parity check on the data to see if what the platter wrote exactly matched what the write request buffer contained. This type of extra check understandably adds more time to the completion length of a write request, but it adds to the integrity of the system.

Data Integrity (Continued)

- Quorum checking ensures that the correct volume group and logical volume information is used after a system crash and reboot.
- Data relocation automatically handles disk read and write errors.

Notes: Data Integrity (Continued)

Quorum checking is the voting that goes on between disks in a volume group to see if a majority of disks exist to form a quorum that will allow the disks in a volume group to become and stay activated. LVM runs many of its commands and strategies based on the having the most current copy of some data. Thus, it needs a method to compare data on two or more disks and figure out which one contains the most current information. This need gives rise to the need of a quorum. If not enough quorums can be found during a varyonvg command, the volume group will not varyon. Additionally, if a disk dies during normal operation and the loss of the disk causes volume group quorum to be lost, then the volume group will notify the user that it is ceasing to allow any more disk i/o to the remaining disks and enforces this by performing a self varyoffvg. However, the user can turn off this quorum check and its actions by telling LVM that it always wants to varyon or stay up regardless of the dependability of the system. Or, the user can force the varyon of a volume group that doesn't have quorum. At this point, the user is responsible for any strange behavior from that volume group.

There are three types of data relocation: internal to the disk, hardware relocate ordered by LVM, and software relocation. Relocation typically occurs when the system fails to perform a read or a write due to physical problems with the disk platter. In some cases, the data I/O request completes, but with warnings. Depending on the type of recovered error, we may be so wary of the success of the next request to that physical location that LVM orders a relocation to be on the safe side. The lowest logical layer of relocation is the one that is internal to the disk. These types of relocations are typical private to the disk and there is no notification to the user that a relocation occurred. The next level up in terms of relocation complexity is a hardware relocation called for by the LVM device driver. This type of relocation will instruct the disk to relocate the data on one physical partition to another portion (reserved) of the disk. The disk takes the data in physical location A and copies it to a reserved portion of the disk, location B. However, after this is complete the LVM device driver will continue to reference physical location A, with the understanding that the disk itself will handle the true i/o to the real location B. The top layer of data relocation is the "soft" relocation handled by the LVM device driver. In this case, the LVM device driver maintains a bad block directory and whenever it receives a request to access a logical location A, the LVM device driver will look up the bad block table and translate it to actually send the request to the disk drive at physical location B. As a default, AIX will not allow the performance of the intermediate level hardware relocation on non-IBM drives. On IBM drives, LVM knows that we're guaranteed the existence of reserved areas for possible hardware relocation (typically 256 blocks per disk). However, on non-IBM drives, IBM cannot predict what spare blocks may or may not exist for possible relocation. So assuming the worst, LVM only allows the internal disk relocation and the top layer software relocation to occur for non-IBM drives.

Performance

Notes: Performance

This page intentionally left blank.

Mirroring Performance

- Parallel mirroring can decrease performance compared to non-mirrored writes.
- Parallel mirroring dramatically increases write performance over sequential mirroring.
- Parallel mirroring generally improves read performance over both non-mirrored reads and reads from sequential mirrors.
- Parallel mirroring on different disks improves performance.
- Parallel mirroring across different adapters improves performance.

Notes: Mirroring Performance

Disk mirroring can improve the read performance of a system, but at a cost to the write performance. Of the two mirroring strategies, parallel and sequential, parallel is the better of the two in terms of disk I/O. In parallel mirroring, when a read request is received, the LVM device driver looks at the queued requests (read and write) and finds the disk with the least number of requests waiting to execute. The user can see how this strategy of finding the shortest wait line would improve the read time. And with mirroring, two independent requests to two different locations can be issued at the same time without causing disk contention, because the requests will be issued to two independent disks. However, with the improvement to the read request as a result of disk mirroring and the multiple identical sources of reads, the LVM disk driver must now perform more writes in order to complete the write request. With mirroring, all disks that make up a mirror are issued write commands which each disk must complete before the LVM device driver considers a write request as complete.

Mirroring on different disks - The default of disk mirroring is that the copies should exist on different disks. This is for performance as well as data integrity. With copies residing on different disks, if one disk is extremely busy, then a read request can be completed the other copy residing on a less busy disk.

Mirroring across different adapters - Another method to improve disk throughput is to mirror the copies across adapters. This will give you a better chance of not only finding a copy on a disk that is least busy, but it will also improve your chances of finding an adapter that is not as busy. LVM does not realize, nor care, that the two disks do not reside on the same adapter. If the copies were on the same adapter, the bottleneck there is still the bottleneck of getting your data through the flow of other data coming from other devices sharing the same adapter card. With multi-adapters, the throughput through the adapter channel should improve.

Striping

- Striping spreads sequential data in a logical volume across multiple disks to increase the read and write rate.
- The stripe width is the number of disks that a logical volume is spread across.
- The stripe length is the number of consecutive bytes that are placed on a single disk before going to the next disk.
- Striping is best when large reads and writes are common on the logical volume.

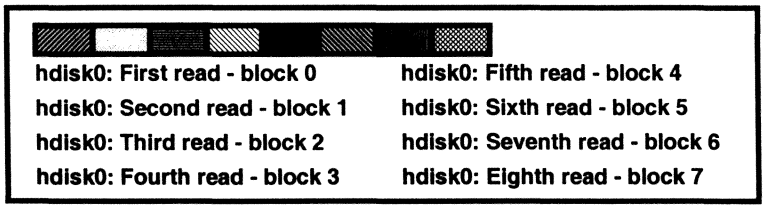
Notes: Striping

Disk striping describes the RAID 0 implementation in software. This functionality is based on the assumption that large amounts of data can be more efficiently retrieved if the request is broken up into smaller requests given to multiple disks. And if the multiple disks are on multiple adapters, then the theory works even better, as mentioned in the previous sections of mirroring across different disks and adapters. In the previous sections, we describe the efficiency gained for mirrors. In this case, the same efficiency is gained with data across disks and adapters, but without mirroring. Thus there is a savings on the write case, as compared to mirrors. But, there is a slight loss in the read case, as compared to mirrors, because now there isn't more than one copy to read from if one disk is busier than the other. Note, you cannot mirror striped logical volumes in AIX.

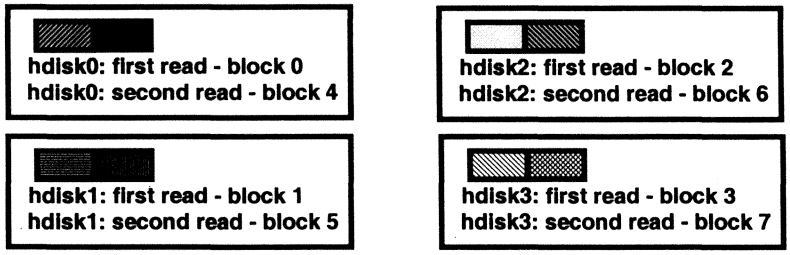
Disk striping is the concept of spreading sequential data across more than one disk to improve disk I/O. The theory is that if you have data that is logically close together, and if you can divide the request into more than one disk I/O, you will reduce the time it takes to get the entire piece of data. This request must be done so it is transparent to the user. The user doesn't know which pieces of the data reside on which disk and does not see the data until all the disk I/O has completed (in the case of a read) and the data has been reassembled for the user. Since LVM has the concept of a logical to physical mapping already built into its design, the concept of disk striping is an easy evolution. Striping is broken down into the "width" of a stripe and the "stripe length". The width is how many disks the sequential data should lay across. The stripe length is how many sequential bytes reside on one disk before the data jumps to another disk to continue the sequential information path.

Striping Example

- **Disk I/O without striping:**



- **Disk I/O with striping:**



Notes: Striping Example

We present an example to show the benefit of striping: A piece of data that is stored of the disk is eight blocks. It takes eight read requests to the same disk to complete the reading of eight blocks. As you can see, since the data is on the same disk, eight sequential reads must be required. If this logical volume were created with a stripe width of 4 (how many disks) and a stripe size of one block (how many data to be read before going to the next disk), then the second diagram (with striping) would hold.

As you can see, each disk only requires two read requests and the time to gather all eight block has been reduced four-fold. However, there is still a bottleneck of having the four independent data disks channel through one adapater card (though a single adapter card can generally handle more data bandwidth than can be generated from one disk). But, this can be remedied with the expensive option of having each disk on an independent adapter card. Note the effect of using striping: the user has now lost the usage of three disks that could have been used for other volume groups.

Performance Tuning

- The "inter-policy" can be specified to spread a logical volume's partition allocation across many disks, as an alternative to striping.
- The "intra-policy" can be specified to place a logical volume's partitions on a disk so as to minimize disk head seek time.
- Turning off mirror write consistency check (MWCC) on a logical volume can improve write performance.
- Turning on write verification on a logical volume can significantly degrade write performance.

Notes: Performance Tuning

There are a few parameters that the user can change per logical volume which will affect the performance of the logical volume in terms of data access efficiency. From experience however, many people have different views of how to achieve that efficiently, so there can't be a specific "right" recommendation given in these notes.

Inter-policy - This comes in two variations, min and max. The two choices tells LVM how the user wishes the logical volume to be spread over the disks in the volume group. With min, this tells LVM that the logical volume should be spread over as few disks as possible. The max policy directs LVM to spread the logical volume over as many disks that are defined by the volume group and limited by the "upper bound" variable. Some users try to use this variation to form a cheap version of disk striping on systems below AIX 4.1. However, it must be stated that the inter-policy is a "recommendation" to the allocp binary, not a strict requirement. In certain cases, depending on what is free on a disk, these allocation policies may not be achievable.

Intra-policy - There are five regions on a disk platter defined by the intra-policy: edge, inner-edge, middle, inner-middle, and center. This policy will tell the LVM what the preferred location of the logical volume on the disk platter. Depending on the value also provided for inter-policy, this preference may or may not be satisfied by LVM. Many users have different ideas as to which portion of the disk is considered the "best", so no recommendation is given in these notes.

Mirror write consistency check - As mentioned before, the mirror write consistency check tracks the last 62 distinct writes to physical partitions. If the user turns this off, they will shorten (although slightly), the path length involved in a disk write. However, the trade-off may be inconsistent mirrors if the system crashes during a write call.

Write verify - This by default is turned off by LVM when a logical volume is created. If this value is turned on for a logical volume, additional time during writes will be accumulated as the IOCC check is performed onto each write to the disk platter.

Problem Determination

Notes: Problem Determination

This page intentionally left blank.

Problem Determination

- **Each problem will depend upon the particular user circumstances.**
- **The problem needs to be isolated to one of three layers first:**
  - ***the middleware layer - JFS, HACMP, database, or an application directly accessing the LVM***
  - ***the LVM itself - high level commands, low level commands, LVM library, or LVM device driver***
  - ***the disk managers - disk device driver, SCSI device driver, SSA device driver, etc.***
- **Determine the exact steps that showed the problem.**
- **If the LVM data structures have been corrupted, determine which LVM data structures are still correct.**

Notes: Problem Determination

Because in most cases, each LVM problem case is specific to a user and their environment, this section isn't a "how-to" section. Instead, it's mostly a checklist section which will help the user gather necessary information to rationally determine the root cause of the problem and if the problem can be fixed in the field, rather than sending to Level 3 software support. And if the problem must be sent to Level 3, this will give suggested information that would speed the problem determination/solution given by Level 3.

root cause:

The first question to be asked is if this problem is really in the LVM layer. The sections that detail how an I/O request is handed down from layer to layer might help clarify all the sections that must be considered.

The most important initial determination is whether the problem is in above the LVM layer, in the LVM layer, or below the LVM layer.

For instance, an application program such as a database or HACMP/6000 that accesses the LVM directly might have a problem. If you can determine what actions these failing programs are attempting to the LVM, then try to recreate this action by hand using a method that is not based on those application programs. If your attempt by hand works, then the focus of the problem shifts "up" to the application program. Obviously if it fails, then you isolated the problem at the LVM layer (or below). Or, the problem could simply be corruption to the data needed by LVM; the programs are behaving correctly, but data needed by LVM is corrupted which is causing LVM to behave strangely.

An additional bonus to the field investigator is the fact that most high-level commands are shell scripts. Thus, if they are familiar with shell programming, they may turn on the shell output and what the execution of the shell commands to observe the failure point. This information might produce additional helpful information to the problem record.

Finally, if there is corruption or loss of LVM data (such as a disk accidently erased from a volume group), it helps to find the exact steps performed (or even not performed) by the user so that the investigator can deduce the state of the system and what useful LVM information is left behind.

Problem Determination

- Try to distill the problem to the simplest case.
- Try to determine what has been lost and what is left behind.
- Try to determine whether sufficient information is still available to allow the situation to be recovered from.
- Sometimes the design of the system prevents error recovery.

Notes: Problem Determination

Many times problem reports from the field to Level 3 concerning LVM are difficult to investigate because clarification is required (to determine the root cause of the problem). Or, the problem is described with the complex user configurations. If it is possible, the most basic action of the LVM is the one that should be investigated. This is not always possible as some problem may only be exposed when running in a complex environment. However, whenever possible one should try to distill the case into how the action to a logical volume is causing misbehavior by the system. And in that clarification, a non-LVM root cause may be discovered instead.

Generally, if some sort of accident has resulted in data corruption or loss of LVM required information, you need to determine what LVM information is still correct. Given the state of the system before the corruption, the steps that most likely caused the corruption, and the current state of the machine, one can deduce what is left to work with. Sometimes one will receive conflicting information. This is because part of the ODM disagrees with part of the VGDA. The ODM is the one that is easily alterable (compared to the VGDA), so it is more likely to have become corrupted.

Sometimes you have enough information to know what is missing and what should be done to repair the system. However, the design of ODM, the system configurator, and LVM prevents the repair. By fixing one problem, another is spawned. And, one is caught in a deadlock situation that cannot be fixed unless one wrote very specific kernel code to repair the internal aspects of the LVM (most likely the VGDA). This is not a trivial solution, but it is possible. It is only through experience that a judgement can be made if recovery can be attempted.

Problem Recovery

- Warn the user of the consequences of attempting to recover from a problem.
- Gather all possible data about the steps that lead up to the situation.
- Save off all data that might be changed by attempting to recover the system.
- Each case is different, so the solution must be tailored to the problem.

Notes: Problem Recovery

Although this might seem a trivial step, when you attempt problem recovery, most of the time you must alter or destroy an important internal structure within the LVM (such as the VGDA). Once this is done, if the recovery attempt didn't work, the user's system is usually in worse shape than before the recovery attempt. Many users will decline the recovery attempt once this warning is given. However, it is better to warn them ahead of time!

While the volume group is still partially accessible, gather all possible data about the current volume group. The VGDA will provide information about missing logical volumes, which will be important. Once the recovery procedure starts, important reference information such as that gathered from the VGDA will be lost for good. And if your information is incomplete, then you may be stuck with no where to go.

Before starting the recovery, make a copy of files that can be restored in case something goes wrong. A good example would be something like the ODM database files that reside in /etc/objrepos. Sometimes the recovery steps involves deleting information from those databases. And once deleted, if one is unsure of their form, one can't try to recreate some of the structures or values.

Also saving off the entire contents of the disk before attempting recovery is a difficult step, but this gives the best likelihood of being able to recover from a failed recovery operation.

Since each LVM problem is most likely going to be unique for that system, these notes cannot provide a list of steps one would take in a repair. Once again, the recovery steps must be based on individual experiences with LVM.

System Dumps

Notes: System Dumps

This module will give the student an understanding of what system dumps are and how they are created.

Outline

- *Dump Causes*
- *How Dumps are Created*
- *The Master Dump Table*
- *Component Dump Tables (CDTs)*
- *New CDTs for 4.3*
- *The Dump Process*
- *Component Dump Routines*
- *Dump Devices*

Notes: Outline

This page intentionally left blank.

Dump Causes

- **Two types: system-initiated and user-initiated**
- **System-initiated dumps have three possible causes:**
  - ***An exception in kernel mode that was unexpected or that we can't handle***
    - ***Page faults while interrupts disabled***
    - ***References to invalid addresses***
    - ***Invalid op codes***
  - ***A trap instruction in kernel code***
    - ***Usually embedded within an ASSERT macro call***
    - ***Indicates an invalid kernel state***
  - ***A call to the* panic() *kernel service***
    - ***Takes a string as a parameter***
    - ***Indicates an invalid kernel state or state that cannot be recovered from***

Notes: Dump Causes

Dumps can be requested by users in several different ways, which we have discussed previously. When we look at how the system decides to generate a dump, though, we see three main causes: an exception while operating in kernel mode that cannot be handled, a trap instruction in kernel code, or a call to the panic() kernel service. There is one common theme here - only kernel code can cause the system to crash and a dump to be generated. The reasons for this should be obvious - we don't want to allow anyone with user access to the system to be able to crash the entire system at will. If an exception occurs while running user code, and the user code can't handle it, then the user process is killed. The same goes for a trap instruction in user code. And since panic() is a kernel service (and not a system call), it can only be called by the kernel and kernel extensions.

How Dumps Are Created

- The kernel steps through the master dump table, looking for dump routines to call
- The dump routines find the data they need and ask the kernel to dump it
- The kernel writes each requested piece of data to the dump device

Notes: How Dumps Are Created

We are now providing an overview of the basic steps in the creation of a dump. We will expand upon each of these steps in the subsequent pages.

The Master Dump Table

- An array based off a pointer in an internal kernel structure (*dmp*)
- Contains pointers to functions that were:
  - *Registered via the* dmp_add *kernel service by kernel extensions*
  - *Pre-loaded by kernel initialization for kernel-specific areas*
- These functions must return a pointer to a component dump table (CDT)

Notes: The Master Dump Table

This page intentionally left blank.

Component Dump Tables (CDTs)

- These are structures of type *struct cdt*
  - *Defined in /usr/include/sys/dump.h*
- Made up of two parts
  - *A fixed-length header (*cdt_head*)*
    - *Contains a magic number, the name of component, the name of kernel extension (if any), and the overall length of the table*
  - *An array of one or more* cdt_entry *structures*
    - *Each one defines a contiguous data area to be dumped*
    - *Each holds a pointer, length, segment register, and name for the data area*

Notes: Component Dump Tables (CDTs)

From /usr/include/sys/dump.h:

```
struct cdt_entry {                      /* component dump table entry */
    char              d_name[8];        /* name of data area to be dumped */
    int               d_len;            /* length of data area to be dumped */
    char             *d_ptr;            /* offset of data area to be dumped */
    union {
            int          _d_segval;     /* segment value for xmem operations*/
            struct xmem *d_xmemp;       /* for non RT/R2 systems */
    } _d;
};

struct cdt_head {
    int  _cdt_magic;
    char _cdt_name[16];                 /* component dump name */
    int  _cdt_len;                      /* length of component dump table */
};
#define DMP_MAGIC  0xEEEEEEEE

struct cdt {                            /* component dump table */
    struct cdt_head  _cdt_head;
    struct cdt_entry  cdt_entry[1]; /* component dump table entries */
};

#define cdt_magic _cdt_head._cdt_magic
#define cdt_name  _cdt_head._cdt_name
#define cdt_len   _cdt_head._cdt_len
```

New CDTs for 4.3

- **AIX 4.3 supports more than 4Gb of real memory**
  - ***Some VMM data is now kept in real memory only***
    - ***No virtual mode access is allowed***
  - ***Real addresses can now require 64 bit pointers***
- **Existing CDT structures require 32-bit virtual addresses**
  - ***So new definitions were needed***

Notes: New CDTs for 4.3

This page intentionally left blank.

New CDTs for 4.3

- New type in sys/dump.h: *struct cdt_vr*
- Also has two parts
  - *Fixed length header as before (*cdt_head*)*
    - *New magic number (*DMP_MAGIC_VR*) identifies this as cdt_vr*
  - *One or more* cdt_entry_vr *structures*
    - *Each defines a contiguous data area to be dumped*
    - *Contains a longer name, the length of the data area, a set of flags detailing the type of address, and either*
      - *A virtual address and corresponding segment register, OR*
      - *A 64-bit real address*

Notes: New CDTs for 4.3

From the 4.3 /usr/include/sys/dump.h:

```
/*
 * cdt_entry_vr is for dumping real memory by virtual or real address.
 * It is only valid for dump tables with the DMP_MAGIC_VR magic number.
 */
struct cdt_entry_vr {                   /* dump table entry for v/r addresses*/
        char          d_name[16];       /* name of data area to be dumped */
        int           d_len;            /* length of data area to be dumped */
        int           d_addrflags;      /* Address flags. */
        union {
                struct {
                    char *_d_ptr;        /* data pointer */
                    int _d_segval;       /* segment id */
                } _d;
                unsigned long long _d_realaddr; /* real address */
        } _u;
};
....
/*
 * Component dump table which may contain real or virtual addresses.
 */
struct cdt_vr {
      struct cdt_head  _cdt_head;
      struct cdt_entry_vr cdt_entry[1];  /* component dump table entries */
};
#define DMP_MAGIC_VR 0xEEEEEEEF
```

The Dump Process

- **Interrupts are disabled**
  - ***Important - we can't have a page fault!***
- **`0c9` or `0c2` are written to the LED display, if present**
- **Header information about the dump is written to the dump device**
- **The kernel steps through each entry in the master dump table, calling each component dump routine twice**
  - ***Once to indicate that the kernel is starting to dump this component***
    - `1` ***is passed as a parameter***
  - ***Again to say that the dump process is complete***
    - `2` ***is passed***

Notes: The Dump Process

If the dump is taken as the result of a system crash (including a *panic*), then `0c9` is displayed in the LEDs when the dump starts, and flashing `888` appears when the dump is finished. If the dump was requested by the user (reset button, sysdumpstart), then `0c2` appears when the dump starts, and `0c0` when it completes.

The Dump Process

- **After the first call to a component dump routine, the kernel processes the CDT that was returned**
  - *For each CDT entry, the kernel*
    - *Checks every page in the identified data area to see if it is in memory or paged out*
    - *Builds a bitmap indicating each page's status*
    - *Writes a header, the bitmap, and those pages which are in memory to the dump device*
  - *Note: this process ensures that the kernel's dump facility will not cause a page fault*
    - *We can't have page faults because interrupts are disabled*
- **Once all dump routines have been called, the kernel enters an infinite loop, displaying `0c0` or flashing `888`**

Notes: The Dump Process

This page intentionally left blank.

Component Dump Routines

- **When called with a 1:**
  - ***Make any necessary preparations for dumping***
    - ***For example, they may read device-specific information from an adapter***
      - ***The FDDI device driver does this***
  - ***Fill in the component dump table***
    - ***Most device drivers do this during their initialization***
  - ***Return the address of the component dump table***
- **When called with a 2:**
  - ***Clean up after themselves***
    - ***In reality, most routines either return immediately, do some debug printfs and then return, or else they ignore the parameter entirely and return the same thing every time***

Notes: Component Dump Routines

A component dump routine may or may not do a lot of work when called with a 1. Many simply return the address of some previously-initialized CDT, but some (for example, the thread table and process table dump routines) actually build the CDT from scratch.

The original rationale for the second call to each dump routine was to provide notification that the dump process had finished with that component's dump data. In practice, however, no one really cares. The routines that just return an address don't even bother to look at the parameter they were passed. The routines that build the data on the fly look for a 2 and return immediately. The most that any routine today does with this second call is to issue some debug printf call. This is generally used to debug the component dump routine itself, by verifying that the system dump facility was able to successfully process its CDT.

Dump Devices

- **Dumps can be written to most devices**
  - ***Raw disks, logical volumes, tape drives, etc.***
- **Any dump device must have a dddump() entry point defined in the device switch table**
  - ***Dump runs disabled for interrupts, so normal disk I/O processing is not available***
    - ***For example, I/O completion interrupts will not be seen***
  - ***dddump is a synchronous write interface***
    - ***If dddump does not return, the entire dump process hangs***
- **Serious hardware problems will prevent successful dumps**
  - ***For example, a hung data bus means we can't write to disk at all***

Notes: Dump Devices

We now switch to a discussion of the devices to which dump data gets written. Any device can be a dump device, as long as it has registered a dddump() entry point in its device switch table entry. This entry point must separate from the ddwrite() entry point, because the two are called under very different circumstances.

ddwrite() is called with interrupts enabled, and usually on behalf of a user program. The standard operating mode here is for the device driver to schedule a write operation to the device, then optionally put the caller to sleep. When the write completes, a completion interrupt (a.k.a. IODONE interrupt) is generated by the device. The device driver's interrupt handler routine is then called. This routine figures out which I/O operation is complete, then wakes up the appropriate thread (if necessary) with some indicator of success or failure for the write. That thread then continues execution.

Of course, this is completely inappropriate when we need to write dump data. The primary consideration is that interrupts must remain completely disabled while dump is in progress. Thus, the dddump() interface is defined to be a completely synchronous interface - a dddump() entry point must not return until the requested write is complete. Rather than use I/O scheduling and completion interrupts, a dddump() entry point generally uses a technique called "polling," where the write is started, and then status indicators are constantly checked until we are sure the write is done. This method is obviously no good for normal device operation - it would have a serious impact on system throughput. But when we are dumping the system, there should be nothing else going on, so we can waste all the CPU cycles we want waiting for dump writes to complete.

Dump Problems

- **Completion code 0c4**
  - ***The dump started, but failed to complete***
    - ***Means that the dump device is not large enough***
    - ***Artificially large dumps can be caused by component dump routines that return corrupted component dump tables***
  - ***Partial dumps <u>can</u> be used for debug***
    - ***The kernel is dumped first***
  - ***Note: In 4.2.0, 4.1.4, and earlier releases, this code was also used to indicate I/O errors and other miscellaneous dump problems***

Notes: Dump Problems

This page intentionally left blank.

Dump Problems

- **Completion code 0c5 - "old" behavior**
  - ***Some "bad" problem happened while dumping***
    - ***Dump device entry point (dddump()) returns an error indication***
    - ***An exception occurs during dump, e.g. a page fault***
  - ***There may be some valid dump data***
    - ***If so, most likely cause is that a component dump routine took a page fault***
- **Code 0c5 - "new" behavior**
  - ***Means that some exception occurred while running the dump-processing code in the kernel***
    - ***Component dump routine errors handled separately (next page)***
    - ***Dump device errors also handled separately***
  - ***Now should be quite rare***

Notes: Dump Problems

Under the old behavior of dump, the 0c5 dump completion code isn't always the disaster it is cracked up to be. The documentation states that this code means that the dump "failed to start." However, there may be valid dump data after all.

This code can be caused by a failure of the dump device itself, which usually means that there is no dump at all. However, it can also be caused by a page fault that occurs while we are dumping. As explained before, though, the kernel's dump routines check all data areas identified in CDT entries to make sure they are in memory before they are read. So, if a page fault does occur while dump is in progress, it is most likely to be caused by a component dump routine.

This problem has been seen, for example, when a device is configured, but not open. Many device drivers register their component dump routines during configuration, but avoid pinning memory until the device is actually opened. When a dump occurs, the kernel has no way to see whether the device is open that corresponds to a particular component dump routine. So, the routine is going to be called during dump. If the routine refers to memory that has not been pinned, the chances are that a page fault will occur.

Since 4.1.5 and 4.2.1, however, this code is much more specific. It means exactly that an exception was raised while the main dump code (not a component dump routine) was running. This should be very rare, and if you see this code, this problem probably belongs to Level 3.

Dump Problems

- **Dump routine failures are now being handled**
  - ***Dump issues a setjmpx() before calling each dump routine***
  - ***If routine fails, dump just skips it***
    - ***Records address of failing dump routine***
    - ***longjmpx() back to continue dump process***
      - ***This handles DSI's, traps, and other dump routine failures***
  - ***After master dump table has been run, a "dump" dump table is dumped***
    - ***Contains the addresses of the routines that failed***
  - ***Available in 4.1.5 and 4.2.1***

Notes: Dump Problems

In 4.1.5 and 4.2.1, dump processing was enhanced to handle one of the primary causes of dump failures - a secondary failure in a component dump routine. Errors in component dump routines used to bring the dump process to an immediate halt. Figuring out which routine caused the failure was often a guessing game, and there was absolutely no information as to why the failure occurred.

Now, through, the dump process uses the setjmpx and longjmpx kernel facilities to set up exception handlers before each dump routine is called. setjmpx is similar in function to the C function setjmp (and longjmpx is similar to longjmp) in that they allow you to return to a previous point of execution no matter how deeply you may be nested in the call stack, or (in this case) no matter how many errors and exceptions we may have pending.

When setjmpx is called, system state is partially saved, and the call returns 0. When a component dump routine takes a DSI or program interrupt, the kernel's exception handler detects that a setjmpx is currently active, so it simply restores the system state to that of when setjmpx was called, then behaves as though setjmpx returned with a non-zero value. The dump code notes the non-zero return, and assumes that the dump routine it was calling failed. The return value is a basic indication of the type of failure (usually EFAULT, a page fault).

The dump process records the failing dump routine, the return code, and then just moves on to the next routine. After all routines have been called, if there were any failures, the dump process adds its records to the end of the dump. Thus, the guesswork has been removed, and the rest of the dump has been preserved.

Programming Examples

Notes: Programming Examples

This page intentionally left blank.

Outline

- **Filesystem Manipulation through System Calls**
  - ***Filesystem manipulation using system calls***
    - ***creat / open / close / read / write / lseek / unlink system calls.***
  - ***Directory data structure***
    - ***mkdir / opendir / readdir / seekdir / closedir / rmdir / chdir system calls.***
    - ***chroot system call.***

Notes: Outline

This page intentionally left blank.

Outline

- **Processes and Threads**
  - ***Processes and threads creation***
    - ***fork / exec / system calls.***
    - ***pthread_create / pthread_attr_init / pthread_attr_setdetachstate system calls.***
  - ***Processes and threads manipulation***
    - ***wait / exit system calls.***
    - ***pthread_attr_destroy / pthread_join / pthread_exit system calls.***

Notes: Outline

This page intentionally left blank.

Objectives

Describe the nature of system calls.

Describe how system calls are different from user level functions.

Describe the difference between *user-mode and kernel-mode.*

Describe the actions of a system call handler.

Understand the errno variable and its relationship to system calls.

Understand the context of a process.

Be familiar with the most common system calls to manipulate files, threads and processes.

Understand filesystem manipulation through the use of system calls.

Understand process creation through system call usage.

Notes: Objectives

This page intentionally left blank.

File Manipulation

Notes: File Manipulation

This page intentionally left blank.

File Manipulation

- **How to create, open, read, and write to files using system calls.**
  - ***Differences between using file descriptor and FILE pointer.***
  - ***creat / open / close / read / write / lseek system calls***
  - ***Manipulating directory information using system calls.***
  - ***mkdir / opendir / readdir / closedir / rmdir / chdir and chroot system calls***

Notes: File Manipulation

File Pointers:

In the Standard I/O Library, a file is called a stream, and is described by a pointer to an object of type FILE, called a file pointer. The FILE data type is defined in the include file <stdio.h>, which should be included before using any of the stdio routines. There are three defined file pointers, stdin, stdout, and stderr. These refer to standard input, standard output, and standard error, respectively.

Opening and Creating Files using FILE Pointers:

In order to read from or write to a file, that file must be opened for reading or writing (or both). The fopen() routine is used for this purpose. fopen() takes two arguments: a character string containing the name of the file to open, and a character string describing how that file should be opened. It returns a pointer to an open stream of type FILE, or the constant NULL if the file could not be opened.

File Descriptors:

When using the low-level interface, a file is referred to using a file descriptor, which is simply a small integer. As with stdio, there are three predefined file descriptors, 0, 1 and 2, which refer to the standard input, standard output, and standard error output respectively.

Opening and Creating Files using File Descriptors:

The open routine is used to open a file for reading and/or writing, or to create it. open() takes three arguments: a character string containing the name of the file to open, an integer specifying how the file is to be open, and an integer mode for use when creating the file. It returns an integer file descriptor on success, or -1 if the file could not be opened.

File Manipulation (File Descriptor)

Refer to *Programming Example 1* at the end of this chapter:

```
18| #include <stdio.h>
19| #include <sys/errno.h>
20| #include <sys/stat.h>
21| #include <fcntl.h>

23| extern int errno;

25| main (int argc, char *argv[])
26| {
27|     struct stat buffer;
28|     int n;
29|     int from, to;
30|     char buf[1024];
```


Notes: File Manipulation (File Descriptor)

Lines 18-21 - header information

Line 23 - declaration to use the global errno variable

Line 27 - structure stat is defined in <sys/stat.h>. This structure is used as a communication method between the program and the stat() system call

Line 29 - "from" is the file descriptor of the file to copy from

Line 29 - "to" is the file descriptor of the file to copy to

Line 30 - buf[ ] is the character array to contain the information read in from the "from" file

File Manipulation (File Descriptor)

```
40|    /* Open the from_file for reading - but first check for */
41|    /* existence of the file */
42|    if ((stat(argv[1], &buffer)) == -1) /* stat failed */
43|    {
44|        if (errno != ENOENT)        /* Because something went wrong */
45|        {
46|            fprintf(stderr, "%s Unable to stat %s\n", argv[0],
47|                    argv[1]);
48|            perror(argv[1]);
49|            fflush(stderr);
50|            exit(errno);
51|        } else {            /* Because file does not exists */
52|            fprintf(stdout, "From file: %s does not exists\n",
53|                    argv[1]);
54|            fflush(stderr);
55|            exit(1);
56|        }
57|    }

58|    if ((from = open(argv[1], O_RDONLY)) < 0)     /* open failed */
59|    {
60|        perror(argv[1]);
61|        exit(errno);
62|    }
```


Notes: File Manipulation (File Descriptor)

Line 42 - Check for existence of the file using the stat system call.

Line 58 - Try to open the file. If open fails, exit with the error number of the failure.

File Manipulation (File Descriptor)

```
66|    if ((stat(argv[2], &buffer)) == -1)    /* stat failed */
67|    {
68|       if (errno != ENOENT)        /* something wrong */
69|       {
70|          fprintf(stderr, "%s: Unable to stat %s\n",
71|                argv[0], argv[2]);
72|          perror(argv[2]);
73|          fflush(stderr);
74|          exit(errno);
75|       } else {        /* File does not exists */
76|          fprintf(stdout,
77|          "Creating %s because it did not exists...",argv[2]);
78|          /* Create the file */
79|          if ((to = creat(argv[2],O_RDWR|0644)) == -1)
80|          {        /* Create failed */
81|             fprintf(stderr, "\nUnable to create %s\n",
82|                   argv[2]);
83|             perror("creat");
84|             fflush(stdout);
85|             exit(errno);
86|          } else        /* Create OK */
87|             fprintf(stdout, "Succeeded.\n");
88|       }
```


Notes: File Manipulation (File Descriptor)

Line 66 - Check for existence of the "to" file using the stat system call.

Line 79 - Try to create the file if it wasn't found.

File Manipulation (File Descriptor)

```
89|     } else {
90|        /* Open the file for write */
91|        fprintf(stdout, "File %s exists. Open it for appending...",
92|                argv[2]);
93|        if ((to = open(argv[2], O_WRONLY|O_APPEND)) == -1)
94|        {
95|           fprintf(stderr, "\nUnable to open %s\n",
96|                   argv[2]);
97|           perror(argv[2]);
98|           fflush(stdout);
99|           fflush(stderr);
100|           exit(errno);
101|        } else
102|           fprintf(stdout, "Succeeded.\n");
103|     }
```


Notes: File Manipulation (File Descriptor)

Line 93 - If "to" file exists, open it for appending.

File Manipulation (File Descriptor)

```
104|    /* Now read a buffer-full at a time from the from-file and */
105|    /* write it to the to-file.  Note that we only write the  */
106|    /* number of characters read in rather than always writing */
107|    /* 1024 characters */
108|
109|    fprintf(stdout, "Copying from %s to %s",argv[1], argv[2]);
110|    while ((n = read(from,buf, sizeof(buf))) > 0)
111|    {
112|       fprintf(stdout, ".");
113|       write (to,buf,n);
114|    }
115|    fprintf(stdout, "Done\n");
116|
117|    /* Now close the files */
118|    fflush(stdout);
119|    close (from);
120|    close (to);
121|    exit(0);
122| }
```


Notes: File Manipulation (File Descriptor)

Lines 109-114 - copy data from the "from" file to the "to" file, one buffer at a time.

File Manipulation (FILE Pointer)

Refer to *Programming Example 2* at the end of this chapter:

Program declaration

```
12| #include <stdio.h>
13| #include <sys/errno.h>
14|
15| extern int errno;
16|
17| main (int argc, char *argv[])
18| {
19|     int c;
20|     FILE *from, *to;
```


Notes: File Manipulation (FILE Pointer)

Line 20 - Declare FILE pointers for use with the fopen() system call.

File Manipulation (FILE Pointer)

```
32|    /* Open the from_file for reading */
33|    if ((from = fopen(argv[1], "r")) == NULL)
34|    {
35|        perror(argv[1]);
36|        exit(errno);
37|    }
38|
39|    /* Open the to_file for appending.  If to_file does */
40|    /* not exists, fopen will create it */
41|    if ((to = fopen(argv[2], "a")) == NULL)
42|    {
43|        perror(argv[2]);
44|        exit(errno);
45|    }
```


Notes: File Manipulation (FILE pointer)

Line 33 - Notice that we use the fopen() system call with the FILE pointer. We also use a different mode format.

File Manipulation (FILE Pointer)

```
46|
47|     /* Now read characters from from_file until we hit */
48|     /* EOF, and put them onto to_file */
49|     while ((c = getc(from)) != EOF)
50|         putc(c,to);
51|
52|     /* Now close the files */
53|     fclose(from);
54|     fclose(to);
55|     exit(0);
56| }
```


Notes: File Manipulation (FILE pointer)

Lines 49-50 - Copy from "from" file to "to" file, one character at a time.

File Manipulation (lseek)

Refer to *Programming Example 4* at the end of this chapter:

Header Information

```
14| #include <stdio.h>
15| #include <sys/errno.h>
16| #include <sys/file.h>
17| #include <fcntl.h>
```

Program Declaration

```
19| struct record {
20|     int uid;
21|     char login[9];
22| };
23|
24| char *logins[] = { "user1", "user2", "user3", "user4", "user5" };
25|
26| void main(void)
27| {
28|     int i, fd;
29|     struct record rec;
```


Notes: File Manipulation (lseek)

Moving Around in Files:

It is often necessary to move to a specific location in a file before reading or writing data.

The low-level routine for moving around in a file is called lseek(). The routine accepts three arguments: a file descriptor to an open file, a long integer specifying the number of bytes to move, called an offset, and an integer indication from where in the file the offset is to be taken. The third argument is usually specified as one of the constant defined in <sys/file.h>.

File Manipulation (lseek)

```
31|    /* Open the data file for writing */
32|    if ((fd = open ("datafile", O_WRONLY|O_CREAT, 0644)) == -1)
33|    {
34|        perror("datafile");
35|        exit(1);
36|    }
```

Create the record and call putrec( ) to put the record into the file

```
38|    /* For each user, going backward... */
39|    for (i=4; i>=0; i--)
40|    {
41|        /* Create the record */
42|        rec.uid = i;
43|        strcpy(rec.login, logins[i]);
44|
45|        /* Output the record. Notice we pass the address of */
46|        /* the structure */
47|        putrec(fd, i, &rec);
48|    }
49|    close(fd);
50|    exit(0);
51| }
```


Notes: File Manipulation (lseek)

Line 32 - Open the file for writing, create it if necessary.

File Manipulation (lseek)

```
60| /* putrec --write the record in the i'th position */
61| putrec(fd, i, r)
62| int i, fd;
63| struct record *r;
64| {
65|     /* Seek to the i'th position from the beginning of the file */
66|     printf("Seek to %d..", lseek(fd, (long)i * sizeof(struct record),
67|       L_SET));
68|
69|     /* Write the record.  We want to write one object the size of
70|     /* a record structure. */
71|     printf("Wrote %d bytes\n", write(fd, (char *)r,
72|     sizeof(struct record)));
73| }
```


Notes: File Manipulation (lseek)

Procedure to put the record into the datafile, using lseek( ) to seek to the right position in the file.

Line 66 - set file offset to the location of the i-th record in the file.

Directory Manipulation

Notes: Directory Manipulation

This page intentionally left blank.

Directory Manipulation

Refer to *Programming Example 4* at the end of this chapter:

Header information

```
15| #include <stdio.h>
16| #include <sys/types.h>
17| #include <dirent.h>
18|
```

Program declaration

```
19| extern int errno;
20| main()
21| {
22|     DIR *dp, *dp1;
23|     struct dirent *dir, *dir1;
24|
```


Notes: Directory Manipulation

Directories provide the mapping between the names of the files and the files themselves, thus inducing a structure on the file system as a whole. A directory contains a number of files; it may also contain subdirectories which in turn contain more files. A directory behaves exactly like an ordinary file when read.

Reading Directories:

A directory structure contains the structures of type dirent, defined in the include file <dirent.h> (once again, <sys/types.h> must be included to define the types used in the structure):

To read a directory, the user declares a pointer of type DIR, similar to stdio's FILE pointer. The opendir() routine opens the directory given as argument and returns a pointer of type NULL if the directory cannot be opened. The readdir() routine returns a pointer to a directory entry, or NULL on end-of-file. The closedir() routine closes the directory file.

Line 22 - Declare dp and dp1 as DIR pointers

Line 23 - Declare dir and dir1 as pointers to structure dirent.

Directory Manipulation

```
25|    if (fork())          /* parent */
26|    {
27|       fprintf(stdout,
28|       "\n\nParent process.  Listing of current dir...\n");
29|       /* Open up the directory */
30|       if ((dp = opendir(".")) == NULL)
31|       {
32|          fprintf(stderr, "cannot open directory.\n");
33|          perror("opendir");
34|          exit(1);
35|       }
36|       /* Read entries... */
37|       while ((dir = readdir(dp)) != NULL)
38|          printf("%s      ", dir->d_name);
39|       closedir(dp);
```


Notes: Directory Manipulation

Create a child process. Each will print out the contents of a different directory. This is the code for the parent process.

Line 30 - Open current directory.

Line 37 - Read directory, one entry at a time.

Directory Manipulation

```
40|     } else {
41|       fprintf(stdout, "\n\nChild process.  Listing of /tmp...\n");
42|       /* Change directory to /tmp */
43|       if (chdir("/tmp") == 0)
44|       {
45|          if ((dp1 = opendir(".")) == NULL)
46|          {
47|             fprintf(stderr, "cannot open directory.\n");
48|             perror("opendir");
49|             exit(1);
50|          }
51|          /* Read entries...*/
52|          while ((dir1 = readdir(dp1)) != NULL)
53|             printf("%s      ", dir1->d_name);
54|          closedir(dp1);
55|       }
56|     }
57|     exit(0);
58| }
```


Notes: Directory Manipulation

This is the code for the child process.

Line 43 - Change current directory to "/tmp".

Line 45 - Open current directory.

Line 52 - Read directory, one entry at a time.

Directory Manipulation

Refer to *Programming Example 5* at the end of this.chapter:

Header information

```
15| #include <stdio.h>
16| #include <sys/types.h>
17| #include <dirent.h>
```

Program Declaration

```
19| extern int errno;
20| main()
21| {
22|     DIR *dp, *dp1;
23|     struct dirent *dir, *dir1;
```


Notes: Directory Manipulation

Changing Current Directory of a Process:

When the system is booted, the first process uses the root directory of the root file system as its current directory. New processes created with the fork subroutine inherit the current directory used by the parent process. The chdir() subroutine changes the current directory of a process. The chdir() subroutine parses the path name to ensure that the target file is a directory and that the process owner has permissions to the directory. After the chdir() subroutine is run, the process uses the new current directory to search all path names that do not begin with a / (slash).

Changing the Root Directory of a Process:

Processes use the global file system root directory for all path names starting with a / (slash). Processes can change their understanding of the root directory through the chroot subroutine.Child processes of the calling process consider the directory indicated by the chroot() subroutine as the logical root directory of the file system. All path name searches beginning with a / (slash) begin at this new root directory.

Directory Manipulation

```
25|    if (fork())          /* parent */
26|    {
27|        fprintf(stdout,"\n\nParent process.\n");
28|        fprintf(stdout, "Changing directory to /tmp/foo...");
29|        if (chdir("/tmp/foo") == -1)
30|        {
31|            fprintf(stderr,
32|                    "Unable to change directory to /tmp/foo\n");
33|            perror("chdir");
34|            exit(1);
35|        } else
36|            fprintf(stdout, "OK\n");
37|        /* Open up the directory */
38|        if ((dp = opendir(".")) == NULL)
39|        {
40|            fprintf(stderr, "cannot open directory.\n");
41|            perror("opendir");
42|            exit(1);
43|        }
44|        /* Read entries... */
45|        while ((dir = readdir(dp)) != NULL)
46|            printf("%s      ", dir->d_name);
47|        printf("\n\n");
48|        closedir(dp);
```


Notes: Directory Manipulation

Use fork() to create a child process. This is the code for the parent process.

Line 29 - Change current directory to "/tmp/foo".

:Line 38 - Open current directory.

Line 45 - Read directory, one entry at a time.

Directory Manipulation

```
49|     } else {
50|         fprintf(stdout, "\n\nChild process.\n");
51|         fprintf(stdout, "Setting new root directory to /tmp...");
52|         if (chroot("/tmp") == -1)
53|         {
54|             fprintf(stderr, "Unable to chroot to /tmp\n");
55|             perror("chroot");
56|             exit(1);
57|         } else
58|             fprintf(stdout, "OK\n");
59|         fprintf(stdout, "Now changing to /tmp/foo with root at /tmp...");
60|         /* Change directory to /tmp */
61|         if (chdir("/tmp/foo") == 0)
62|         {
63|             fprintf(stdout, "OK\n");
64|             if ((dp1 = opendir(".")) == NULL)
65|             {
66|                 fprintf(stderr, "cannot open directory.\n");
67|                 perror("opendir");
68|                 exit(1);
69|             }
```


Notes: Directory Manipulation

This is the code for the child process.

Line 52 - Change root directory to "/tmp".

Line 61 - Change current directory to "/tmp/foo". (This is really "/tmp/tmp/foo".)

Line 64 - Read directory, one entry at a time.

Directory Manipulation

```
70|             /* Read entries...*/
71|             while ((dir1 = readdir(dp1)) != NULL)
72|                 printf("%s      ", dir1->d_name);
73|             closedir(dp1);
74|             printf("\n\n");
75|         } else {
76|             fprintf(stderr, "Failed\n");
77|             perror("chdir");
78|             exit(1);
79|         }
80|     }
81|     exit(0);
82| }
```


Notes: Directory Manipulation

Continuation of the listing of the child process code.

Processes and Threads

Notes: Processes and Threads

This page intentionally left blank.

Processes and Threads

- **Process creation and manipulation**
  - ***Process creation***
    - ***fork / exec system calls.***
  - ***Process manipulation***
    - ***wait / exit system calls.***

Notes: Process Creation and Manipulation

Process Control:

There are two reasons why UNIX applications use fork( ):

To create a new thread of control within the same program. This use can be replaced by pthread_create( ), which creates a new thread in a multi-threaded process. (Note that there is no requirement to change applications to use threads rather than multiple processes; however, it may be more efficient to do so.)

To create a new process running a different program; the call to fork( ) is then soon followed by a call to exec( ).

In a multi-threaded application, the semantic of fork( ) could be either to copy all of the threads into the new process or to copy only the thread that calls fork( ). AIX copies only the thread that calls fork().

Fork( ), exec( ), wait( ) and exit( ):

In AIX Version 4.1, the child process is created with a single thread (the calling thread). This new process contains a replica of the calling thread and its entire address space at the time of the fork( ). When a process issues a fork( ) system call, the original process is called the parent process and the newly-created process is called the child process. The following list highlights some of the similarities and differences between these two processes:

Both have different process identification number (PID).

Each process can determine whether it is the parent or the child by the value it receives from fork( ). The child receives 0 and the parent receives the PID of the child process.

Both processes share open files.

Each fork( ) system call increments the total number of processes. A parent process may or may not wait for any of its children to terminate. The various form of the exec system call cause processes to overlay the information they contain with new information. During an exec, the calling process exchanges its current user text and process private segments for new user text and process private segments. After the exec system call, the PID is unchanged and open files remain open (except close-on-exec files).

Process Creation and Manipulation

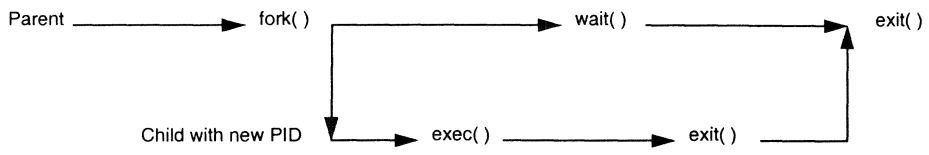


Figure S1 - Parent and Child Relationship

Notes: Process Creation and Manipulation

The exit( ) system call terminates the process that calls it, closes files accessed by the process, notifies the waiting parent (if any), and writes an accounting record, if accounting is enabled. The parent process then clears the entry of the terminated child process from the process table. If the parent was itself terminated, and so cannot fulfill its responsibility to clear the process table entry, the init process becomes the parent process and clears the entry. A zombie process is a terminated process whose entry remains in the process table.

The wait( ) system call suspends the calling process until a child process exits, a child process is stopped in trace mode (the child is traced by its parent), or until the caller receives a signal. A wait( ) system call passes termination status to the parent process, 1 byte (high) as set by exit( ) and 1 byte (low) of system status. This system call also removes zombies from the process table.

The following scenario illustrates the sequence of system calls that synchronize a typical parent process / child process relationship in an interactive environment.

Process <P> executes a fork( ) system call, creating a new process, process <C>.

Process <P> executes a wait( ) system call, and waits for process <C> to finish.

Process <C> executes the exec( ) system call, overlaying itself with the required program.

Process <C> finishes its work and terminates by executing and exit( ) system call.

Process <P> determine the exit status of process <C> through the termination status, then removes the entry for process <C> from the process table and exits.

Process Manipulation

Refer for *Programming Example 6* at the end of this chapter:

Header Information

```
16| #include <stdio.h>
17| #include <unistd.h>
18| #include <sys/errno.h>
19| #include <sys/stat.h>
20| #include <sys/wait.h>
```

We fork( ) a child process and then overlay the image with another executable

```
46|    /* Fork a child process to overlay the exec image */
47|    if (fork() == 0)                /* Child */
48|    {
49|       printf("%s: Child process PID is %d\n",
50|              argv[0],getpid());
51|       printf("%s: Running %s...\n",argv[0],argv[1]);
52|       if ((system(argv[1]) == -1))
53|       {
54|          printf("%s: system failed\n");
55|          perror("system");
56|          exit(-1);
57|       } else
58|       {
59|          printf(" OK\n");
60|          exit(0);
61|       }
```


Notes: Process Manipulation

Line 47 - Fork to create the child process.

Line 52 - The child process executes another application.

Process Manipulation

```
62|     } else                /* Parent */
63|     {
64|         printf("%s: Parent process PID is %d\n",
65|                 argv[0],getpid());
66|         wait(&status);
67|         if (WIFEXITED(status))
68|             printf("%s: Child process exited with %d\n",
69|                     argv[0],WEXITSTATUS(status));
70|     }
71| }
```


Notes: Process Manipulation

Line 66 - The parent process waits for child termination.

Processes Properties

- traditional process attributes
- common address space
- common system resources

Notes: Processes Properties

Threads and Processes:

In traditional single-threaded process systems, a process has a set of properties. In multi-threaded systems, these properties are divided between processes and threads.

Process Properties:

A process in a multi-threaded system is the changeable entity. It must be considered as an execution frame. It has all traditional process attributes, such as:

Process ID, process group ID, user ID, and group ID

Environment

Current directory and root directory

A process also provides a common address space and common system resources:

File descriptors

Signal actions

Shared libraries

Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Thread Properties

- stack
- scheduling properties
- pending and blocked signals

Notes: Threads Properties

Thread Properties:

A thread is the schedulable entity. It has only those properties that are required to ensure its independent flow of control. These include the following properties:

Stack

Scheduling properties (such as policy or priority)

Set of pending and blocked signals

Some thread-specific data.

An example of thread-specific data is the error indicator, errno. In multi-threaded systems, errno is no longer a global variable, but usually a subroutine returning a thread-specific errno value. Some other systems may provide other implementations of errno.

Threads within a process must not be considered as a group of processes. All threads share the same address space. This means that two pointers having the same value in two threads refer to the same data. Also, if any thread changes one of the shared system resources, all threads within the process are affected. For example, if a thread closes a file, the file is closed for all threads.

Threads Terminology

- kernel thread
- user thread
- virtual processor
- process
- multithreaded and monothreaded
- standard UNIX process
- attached and bound
- contention scope: global and local

Notes: Threads Terminology

Terminology:

Speaking about threads may leads to some confusion, according to whether we speak about the kernel (kernel space) or the library (user space). So we need to define some terms used:

A **kernel thread** is the support of execution in the kernel space. It could be either a representation in the kernel of a user thread, or a "pure" kernel thread, not related to a user process.

A **user thread** is one support of execution in user space.

A **Virtual Processor (VP)** may be thought of as the representation of a kernel thread in the library. A user thread needs a VP to execute.

A **process** is an environment of execution. It may be **multithreaded** if it has created more than one thread. It may be **monothreaded** if it has only one user thread (the default at creation time). In this case, it is equivalent to a standard UNIX process.

A **standard UNIX process**, not linked with libpthreads.a, has no user thread and no VP, but always needs a kernel thread to execute.

A thread is said to be **attached** to a VP it a VP is currently running it, from the library point of view. A thread is said to be **bound** to a VP if the thread is attached to it for its whole execution.

The **contention scope** of a thread for a (virtual) process is said to be **local** if the thread contends with other threads of the process to get an available VP. It will be **global** if the thread is **bound** to its VP and then contends only with other threads inside the kernel to get an available (real) CPU.

Threads Manipulation

- thread creation
- waiting for thread termination

Notes: Threads Manipulation

Thread Creation:

The POSIX routine for thread creation is pthread_create( ). A start_routine accepting one argument is designated and a thread ID is returned. Creating a thread is accomplished by calling this routine. This subroutine creates a new thread and makes it runnable.

When calling the pthread_create subroutine, you must specify an entry-point routine. This routine, provided by your program, is similar to the main routine for the process. It is the first user routine executed by the new thread. When the thread returns from this routine, the thread is automatically terminated.

The entry-point routine has one parameter, a void pointer, specified when calling the pthread_create subroutine. You may specify a pointer to some data, such as a string or a structure. The creating thread (the one calling the pthread_create subroutine) and the created thread must agree upon the actual type of this pointer.The entry-point routine returns a void pointer. After the thread termination, this pointer is stored by the threads library unless the thread is detached.

The thread ID of a newly created thread is returned to the creating thread through the thread parameter. The current thread ID is returned by the pthread_self subroutine. A thread ID is an opaque object; its type is pthread_t. In AIX, the pthread_t data type is a pointer. On other systems, it may be a structure, an integer value, or any other data type.

To enhance the portability of programs using the threads library, the thread ID should always be handled as an opaque object. For this reason, thread IDs should be compared using the pthread_equal subroutine. Never use the C equality operator (==), because the pthread_t data type may be neither an arithmetic type nor a pointer.

Waiting for Thread Termination:

The POSIX defined pthread_join( ) function suspends execution of the calling thread until the target thread terminates. The target thread must not be in a detached state. When the function returns, the target thread will be deleted.

Threads Manipulation

- detaching a thread
- thread termination

Notes: Threads Manipulation

Detaching a Thread:

POSIX introduced a pthread_detach( ) function to indicate that storage for that thread can be reclaimed when the thread terminates: At exit, the thread is put on a free list. A thread can be created in the "detach" state if the detach state attribute has been set. At exit time, if a thread is not detached, its resources are kept.

Thread Termination:

pthread_exit( ) is used to terminate a thread. All resources are marked to be freed. If threads are joining this thread, we must use pthread_cond_broadcast( ) on the special condition variable to awake the awaiting threads.

When the function to execute at thread_creation (start_routine) returns, an implicit call to pthread_exit( ) is made with an exit status equal to its return value, except for the initial thread. When the initial thread returns from main, an implicit call to exit( ) is made with an exit status equal to the return value of main.

Building Multithreaded Applications

- compilation commands
- libraries used

Notes: Building Multithreaded Applications

Compiler Invocation:

When compiling a multi-threaded program, you should invoke the C compiler using one of the following commands:

xlc_r Invokes the compiler with default language level of ansi.

cc_r Invokes the compiler with default language level of extended.

These commands ensure that the adequate options and libraries are used.

The following libraries are automatically linked with your program when using these commands:

libc.a Standard C library

libc_r.a Reentrant C library

libpthreads.a Threads library.

Thread Creation

Refer to *Programming Example 7* at the end of this chapter:

```
16| #include <pthread.h>    /* include file for pthreads - the 1st */
17| #include <stdio.h>      /* include file for printf()          */
18| #include <unistd.h>     /* include file for sleep()           */

19| void *Thread(void *string)
20| {
21|       while (1)
22|             printf("%sn", (char *)string);
23|       pthread_exit(NULL);
24| }

25| int main()
26| {
27|       char *e_str = "Thread1!";
28|       char *f_str = "Thread2!";
29|       int rc;
30|
31|
32|       pthread_t e_th;
33|       pthread_t f_th;
```


Notes: Thread Creation

Lines 16 - 18: Header Information

Lines 19 - 24: Procedure to print ouf the message

Lines 25 - 33: Program Declaration

Thread Creation

```
36|     /* Create a thread */
37|     rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
38|     if (rc)
39|         exit(-1);
40|
41|     /* Create another thread */
42|     rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
43|     if (rc)
44|         exit(-1);
45|     sleep(5);
46|
47|     exit(0);
48| }
```


Notes: Thread Creation

Line 37 - Create "e" thread.

Line 42 - Create "f" thread.

Thread Manipulation

Refer to *Programming Example 8* at the end of this chapter:

Header Information

```
14| #include <pthread.h>    /* include file for pthreads - the 1st */
15| #include <stdio.h>      /* include file for printf()          */
```

Procedure to print out the message

```
16| void *Thread(void *string)
17| {
18|     int i;
19|
20|
21|     /* writes five messages and exits */
22|     for (i=0; i<5; i++)
23|         printf("Message #%d %sn", i,(char *)string);
24|     pthread_exit(NULL);
25| }
```


Notes: Thread Creation

This page intentionally left blank.

Thread Manipulation

```
26| int main()
27| {
28|     char *e_str = "Thread1!";
29|     char *f_str = "Thread2!";
30|     int rc;
31|
32|     pthread_attr_t attr;
33|     pthread_t e_th;
34|     pthread_t f_th;

36|     /* creates the right attribute */
37|     pthread_attr_init(&attr);
38|
39|     /* set the value of the detachstate attribute */
40|     /* setting it to PTHREAD_CREATE_UNDETACHED so that when the */
41|     /* thread terminates, its storage is NOT reclaimed by the */
42|     /* system */
43|     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);
```


Notes: Thread Manipulation

Lines 26 - 34: Main routine declarations

Lines 36 - 43: Create the correct attribute and set attribute to PTHREAD_CREATE_UNDETACHED

Thread Manipulation

```
45|    /* creates both threads */
46|    rc = pthread_create(&e_th, &attr, Thread, (void *)e_str);
47|    if (rc)
48|        exit(-1);
49|    rc = pthread_create(&f_th, &attr, Thread, (void *)f_str);
50|    if (rc)
51|        exit(-1);


53|    /* Since we set the attribure to PTHREAD_CREATE_UNDETACHED */
54|    /* earlier, we MUST now destroy it */
55|    pthread_attr_destroy(&attr);
56|
57|    /* joins the threads */
58|    pthread_join(e_th, NULL);
59|    pthread_join(f_th, NULL);
60|
61|
62|    /* The Proper way to exit out of a multithreaded program */
63|    pthread_exit(NULL);
64| }
```


Notes: Thread Manipulation

Lines 45 - 51: Create the threads.

Lines 53 - 64: Reclaim thre threads resources and join them together before exiting.

Lab Exercise 1

Use the crash command to examine threads and processes on the system. To run crash, login to the machine as root and type "crash" at the command line. For a list of crash commands type a "?" and carriage return.

You should see output similar this this:

```
> ?
Dump Analyzer
USAGE: crash [-a] [-i includefile] [systemimagefile [kernelfile]]
Available commands:

buffer  buffer data
buf     buffer headers
cm      change current segment map, used by od
cpu     set current cpu
dblock  STREAMS data block headers
dlock   search for deadlocks
dmodsw  STREAMS driver switch table
ds      data address namelist search
du      dump user area of process
```



```
dump    list components of a dump
errpt   error log messages
file    file table
fmodsw  STREAMS module switch table
fs      formatted kernel stack dump
inode   inode table
inode - inode table with block addresses
knlist  name search from active kernel's knlist
kfp     frame pointer for kernel stack trace start
le      load list entries
linkblk STREAMS linkblk table
mblock  STREAMS message block headers
mbuf    mbuf structures
mlist   list mbuf and what it points to
mst     mst structures
ndb     network debugger
nm      name search
od      dump symbol values
ppd     prints the Per Processor Data Areas
print   formatted structure dump
proc    process table
```

```
proc -  process table long listing
proc -r runnable processes only
!       escape to shell
?       print this list of available commands
qrun    scheduled STREAMS queues
queue   STREAMS write queue
quit    exit
rqueue  STREAMS read queue
socket  socket structures
stack   kernel stack
stat    dump statistics
status  processor status
stream  stream head table
symptom Generate a symptom string
tcb     thread control block
thread  display thread table
thread -        thread table long listing
thread -r       display runable threads
thread -p       display threads of specified process
callout active trb structures
trace   kernel stack trace
```



```
trace -k        trace with extended information
trace -m        trace using mst save area(s)
trace -r        trace using saved kernel frame pointer
ts      text address namelist search
tty     tty structures
user    user area
var     system variables
vfs     virtual file systems
vfs -   virtual file systems + vnodes per vfs
vnode   vnode (virtual inode)
xmalloc kernel xmalloc usage
```

To find the currently running thread and process, use the "status" command.

You should see output similar to this:

```
> status
CPU    TID  TSLOT    PID  PSLOT  PROC_NAME
 0   1ddd    29   1cd4    28  crash
```


To get a list of the processes currently running on the system, use the "p" command.

You should see output similar to this:

```
> p
SLT ST   PID  PPID  PGRP   UID  EUID  TCNT  NAME
 0 a     0     0     0     0     0     1  swapper
     FLAGS: swapped_in no_swap fixed_pri kproc
 1 a     1     0     0     0     0     1  init
     FLAGS: swapped_in no_swap execed
 2 a   204     0     0     0     0     1  wait
     FLAGS: swapped_in no_swap fixed_pri kproc
 3 a   306     0     0     0     0     1  netm
     FLAGS: swapped_in no_swap fixed_pri kproc
 4 a   408     0     0     0     0     5  gil
     FLAGS: swapped_in no_swap fixed_pri kproc
 7 a   754     0   54a     0     0     1  lvmb
     FLAGS: swapped_in kproc orphanpgrp
 8 a   8b2     1   6b4     0     0     1  syncd
     FLAGS: swapped_in orphanpgrp execed
 9 a   9fe     1   9fe     0     0     1  srcmstr
```

```
     FLAGS: swapped_in orphanpgrp execed
10 a    a28    1   a28    0    0    1  cron
     FLAGS: swapped_in orphanpgrp execed
11 a    bee    1   bee    0    0    1  ksh
     FLAGS: swapped_in orphanpgrp execed
12 a    cba    1   cba    0    0    1  errdemon
     FLAGS: swapped_in orphanpgrp signochld
13 a    d16   9fe   d16    0    0    1  sendmail
     FLAGS: swapped_in orphanpgrp execed
14 a    e10   9fe   e10    0    0    1  syslogd
     FLAGS: swapped_in orphanpgrp execed
15 a    f20   9fe   f20    0    0    1  portmap
     FLAGS: swapped_in orphanpgrp execed
16 a   1026   9fe  1026    0    0    1  inetd
     FLAGS: swapped_in orphanpgrp execed
17 a   11fc    1   11fc    0    0    1  uprintfd
     FLAGS: swapped_in orphanpgrp signochld execed
18 a   1202    1   1202    0    0    1  getty
     FLAGS: swapped_in orphanpgrp execed
19 a   13f4   9fe  13f4    0    0    1  qdaemon
     FLAGS: swapped_in orphanpgrp execed
```



```
20 a   140a    1   15fe    0    0    1  diagd
     FLAGS: swapped_in orphanpgrp
22 a   16cc   bee  16cc    0    0    1  script
     FLAGS: swapped_in execed
23 a   17fa   9fe  17fa    0    0    1  writesrv
     FLAGS: swapped_in orphanpgrp execed
24 a   18ce  16cc  16cc    0    0    1  script
     FLAGS: swapped_in
26 a   1ad0  18ce  1ad0    0    0    1  ksh
     FLAGS: swapped_in orphanpgrp execed
28 a   1cd4  1ad0  1cd4    0    0    1  crash
     FLAGS: swapped_in execed
```

To see a list of the currently running threads, use the "th" command.

You should see output similar to this:

```
> th
SLT ST   TID    PID   CPUID POLICY PRI CPU   EVENT PROCNAME   FLAGS
 0 s     3      0      0   FIFO  10 78          swapper
     t_flags: wakeonsig kthread
 1 s    1bb     1      0  other  3c  0           init
     t_flags: wakeonsig
 2 r    205     204    0   FIFO  7f  78          wait
     t_flags: sigslih kthread
 3 s    307     306    0    RR   24  0           netm
     t_flags: sigslih kthread
 4 s    409     408    0    RR   25  0           gil
     t_flags: sigslih kthread
 5 s    50b     408    0    RR   25  0 0015de9c     gil
     t_flags: sigslih kthread
 6 s    60d     408    0    RR   25  0 0015de9c     gil
     t_flags: sigslih kthread
 7 s    70f     408    0    RR   25  0 0015de9c     gil
```



```
     t_flags: sigslih kthread
 8 s    811     408    0    RR   25  0 0015de9c     gil
     t_flags: sigslih kthread
 9 s    907     9fe    0  other  3c  0           srcmstr
     t_flags: wakeonsig
              sel
10 s    abb     8b2    0  other  3c  0 05a5cf58     syncd
     t_flags: wakeonsig
11 s    b5d     754    0  other  3c  0           lvmb
     t_flags: sigslih kthread
12 s    c31     a28    0  other  3c  0 05494ccc     cron
     t_flags: wakeonsig
13 s    df7     bee    0  other  3c  0           ksh
     t_flags: wakeonsig
14 s    e19     e10    0  other  3c  0           syslogd
     t_flags: wakeonsig
              sel
15 s    fc3     cba    0  other  3c  0 0008d298  errdemon
     t_flags: wakeonsig
16 s   1029     f20    0  other  3c  0           portmap
     t_flags: wakeonsig
```

```
            sel
17 s   111f    d16      0  other  3c  0          sendmail
    t_flags: wakeonsig
            sel
18 s   122f    1026     0  other  3c  0            inetd
    t_flags: wakeonsig
            sel
19 s   1305    11fc     0  other  3c  0 0015ddc8  uprintfd
    t_flags: wakeonsig
20 s   140b    1202     0  other  3c  0 05a95a44    getty
    t_flags: wakeonsig
21 s   15fd    13f4     0  other  3c  0 0183f868   qdaemon
    t_flags: wakeonsig
22 s   1613    140a     0  other  3c  0            diagd
    t_flags: wakeonsig
23 s   17d5    16cc     0  other  3c  0 05a96e44    script
    t_flags: wakeonsig
24 s   18d7    18ce     0  other  3d  2 05a96e54    script
    t_flags: wakeonsig
25 s   1903    17fa     0  other  3c  0 00262d8c  writesrv
    t_flags: wakeonsig
```



```
27 s   1bd9    1ad0     0  other  3c  0             ksh
    t_flags: wakeonsig
29 r   1ddd    1cd4     0  other  48  18            crash
    t_flags:
```

To see a list much more detail about each process, use the "p -" command.

You should see output similar to this:

```
> p -
SLT ST   PID  PPID  PGRP  UID EUID TCNT NAME
 0 a     0     0     0     0    0    1 swapper
      FLAGS: swapped_in no_swap fixed_pri kproc

Links: *child:0xe3000100 *siblings:0x00000000 *uidl:0xe3001700
   *ganchor:0x00000000 *pgrpl:0x00000000 *ttyl:0x00000000
Dispatch Fields: pevent:0x00000000 *synch:0xffffffff
   lock:0x00000000 lock_d:0x00000000
Thread Fields: *threadlist:0xe6000000 threadcount:1
   active:1 suspended:0 local:0 terminating:0
Scheduler Fields:  fixed pri: 16 repage:0x00000000 scount:0 sched_pri:127
   *sched_next:0x00000000 *sched_back:0x00000000 cpticks:855
   msgcnt:0   majfltsec:0
Misc: adspace:0x00000808 kstackseg:0x007fffff xstat:0x0000
   *p_ipc:0x00000000 *p_dblist:0x00000000 *p_dbnext:0x00000000
Signal Information:
```



```
   pending:hi 0x00000000,lo 0x00000000
   sigcatch:hi 0x00000000,lo 0x00000000 sigignore:hi 0xffffffff,lo 0xfff7ffff
Statistics: size:0x00000000(pages) audit:0x00000000
   pctcpu:0   minflt:322   majflt:7
SLT ST   PID  PPID  PGRP  UID EUID TCNT NAME
 1 a     1     0     0     0    0    1 init
      FLAGS: swapped_in no_swap execed

Links: *child:0xe3001400 *siblings:0x00000000 *uidl:0xe3000100
   *ganchor:0x00000000 *pgrpl:0x00000000 *ttyl:0x00000000
Dispatch Fields: pevent:0x00000000 *synch:0xffffffff
   lock:0x00000000 lock_d:0x00000000
Thread Fields: *threadlist:0xe60000c0 threadcount:1
   active:1 suspended:0 local:0 terminating:0
Scheduler Fields:  nice: 20 repage:0x00000000 scount:0 sched_pri:127
   *sched_next:0x00000000 *sched_back:0x00000000 cpticks:0
   msgcnt:0   majfltsec:112
Misc: adspace:0x00000505 kstackseg:0x007fffff xstat:0x0000
   *p_ipc:0x00000000 *p_dblist:0x00000000 *p_dbnext:0x00000000
Signal Information:
   pending:hi 0x00000000,lo 0x00000000
```

```
   sigcatch:hi 0x00000001,lo 0x18783eff  sigignore:hi 0x7ffffffe,lo 0xe786c000
Statistics:  size:0x00000019(pages)  audit:0x00000000
   pctcpu:10    minflt:392    majflt:112
SLT ST    PID   PPID   PGRP   UID  EUID  TCNT  NAME
 2 a    204     0     0     0     0     1  wait
      FLAGS: swapped_in no_swap fixed_pri kproc

Links:  *child:0x00000000  *siblings:0x00000000  *uidl:0xe3000200
   *ganchor:0x00000000  *pgrpl:0x00000000  *ttyl:0x00000000
Dispatch Fields:  pevent:0x00000000  *synch:0xffffffff
   lock:0x00000000  lock_d:0x00000000
Thread Fields:  *threadlist:0xe6000180  threadcount:1
   active:1  suspended:0  local:0   terminating:0
Scheduler Fields:   fixed pri:127  repage:0x00000000  scount:0  sched_pri:127
   *sched_next:0x00000000  *sched_back:0x00000000 cpticks:28325
   msgcnt:0    majfltsec:0
Misc:  adspace:0x00000909  kstackseg:0x007fffff  xstat:0x0000
   *p_ipc:0x00000000  *p_dblist:0x00000000  *p_dbnext:0x00000000
Signal Information:
   pending:hi 0x00000000,lo 0x00000100
   sigcatch:hi 0x00000000,lo 0x00000000  sigignore:hi 0x00000000,lo 0x00000000
```



```
Statistics:  size:0x00000000(pages)  audit:0x00000000
   pctcpu:0    minflt:0    majflt:0
SLT ST    PID   PPID   PGRP   UID  EUID  TCNT  NAME
 3 a    306     0     0     0     0     1  netm
      FLAGS: swapped_in no_swap fixed_pri kproc

Links:  *child:0x00000000  *siblings:0xe3000200  *uidl:0xe3000300
   *ganchor:0x00000000  *pgrpl:0x00000000  *ttyl:0x00000000
Dispatch Fields:  pevent:0x00000000  *synch:0xffffffff
   lock:0x00000000  lock_d:0x00000000
Thread Fields:  *threadlist:0xe6000240  threadcount:1
   active:1  suspended:0  local:0   terminating:0
Scheduler Fields:   fixed pri: 36  repage:0x00000000  scount:0  sched_pri:127
   *sched_next:0x00000000  *sched_back:0x00000000 cpticks:0
   msgcnt:0    majfltsec:0
Misc:  adspace:0x00000606  kstackseg:0x007fffff  xstat:0x0000
   *p_ipc:0x00000000  *p_dblist:0x00000000  *p_dbnext:0x00000000
Signal Information:
   pending:hi 0x00000000,lo 0x00000100
   sigcatch:hi 0x00000000,lo 0x00000000  sigignore:hi 0x00000000,lo 0x00000000
Statistics:  size:0x00000000(pages)  audit:0x00000000
```

```
  pctcpu:10    minflt:14    majflt:0
```


To see a list much more detail about each thread, use the "th -" command.

You should see output similar to this:

```
> th -
SLT ST    TID     PID    CPUID  POLICY PRI CPU    EVENT  PROCNAME    FLAGS
 0 s      3       0       0    FIFO  10  78           swapper
      t_flags:  wakeonsig kthread

Links:  *procp:0xe3000000  *uthreadp:0x2ff3b400  *userp:0x2ff3b6e0
    *prevthread:0xe6000000  *nextthread:0xe6000000,  *stackp:0x00000000
    *wchan1(real):0x00000000  *wchan2(VMM):0x00000000 *swchan:0x00000000
    wchan1sid:0x00000000  wchan1offset:0x00000000
    pevent:0x00000000  wevent:0x00000001  *slist:0x00000000
Dispatch Fields:  *prior:0xe6000000  *next:0xe6000000
    polevel:0x0000000b  ticks:0x0b4c  *synch:0xffffffff  result:0x00000000
    *eventlst:0x00000000  *wchan(hashed):0x00000000  suspend:0x0001
    thread waiting for:  event(s)
Scheduler Fields:  cpuid:0x0000  scpuid:0x0000  pri: 16  policy:FIFO
    affinity:0x0000  cpu:0x0078    lpri:  0  wpri:127    time:0x00  sav_pri:0x10
```

```
Misc: lockcount:0x00000000 ulock:0x00000000 *graphics:0x00000000
   dispct:0x00011835 fpuct:0x00000001 *cancel:0x00000000 boosted:0x0000
Signal Information: cursig:0x00 *scp:0x00000001
   pending:hi 0x00000000,lo 0x00000000 sigmask:hi 0x00000000,lo 0x00000000


SLT ST   TID    PID   CPUID POLICY PRI CPU   EVENT PROCNAME   FLAGS
 1 s   1bb      1      0  other 3c  0           init
     t_flags: wakeonsig

Links: *procp:0xe3000100 *uthreadp:0x2ff3b400 *userp:0x2ff3b6e0
   *prevthread:0xe60000c0 *nextthread:0xe60000c0, *stackp:0x2ff22db0
   *wchan1(real):0x00000000 *wchan2(VMM):0x00000000 *swchan:0x00000000
   wchan1sid:0x00000000 wchan1offset:0x0626dcf8
   pevent:0x00000000 wevent:0x00000004 *slist:0x00000000
Dispatch Fields: *prior:0xe60000c0 *next:0xe60000c0
   polevel:0x000000b6 ticks:0x0000 *synch:0xffffffff result:0x00000000
   *eventlst:0x00000000 *wchan(hashed):0x00000000 suspend:0x0001
   thread waiting for: event(s)
Scheduler Fields: cpuid:0x0000 scpuid:0x0000 pri: 60 policy:other
   affinity:0x0000 cpu:0x0000   lpri: 61 wpri:127   time:0xff sav_pri:0x3c
```



```
Misc: lockcount:0x00000000 ulock:0x00000000 *graphics:0x00000000
   dispct:0x00000e22 fpuct:0x00000000 *cancel:0x00000000 boosted:0x0000
Signal Information: cursig:0x00 *scp:0x00000001
   pending:hi 0x00000000,lo 0x00000000 sigmask:hi 0x00000000,lo 0x00000000


SLT ST   TID    PID   CPUID POLICY PRI CPU   EVENT PROCNAME   FLAGS
 2 r   205    204      0   FIFO 7f  78          wait
     t_flags: sigslih kthread

Links: *procp:0xe3000200 *uthreadp:0x2ff3b400 *userp:0x2ff3b6e0
   *prevthread:0xe6000180 *nextthread:0xe6000180, *stackp:0x00000000
   *wchan1(real):0x00000000 *wchan2(VMM):0x00000000 *swchan:0x00000000
   wchan1sid:0x00000000 wchan1offset:0x00000000
   pevent:0x00000000 wevent:0x00000000 *slist:0x00000000
Dispatch Fields: *prior:0xe6000180 *next:0xe6000180
   polevel:0x00000000 ticks:0xd8e3 *synch:0xffffffff result:0x00000000
   *eventlst:0x00000000 *wchan(hashed):0x00000000 suspend:0x0001
   thread waiting for: cpu (ready)
Scheduler Fields: cpuid:0x0000 scpuid:0x0000 pri:127 policy:FIFO
```

```
  affinity:0x0000 cpu:0x0078   lpri:  0 wpri:127   time:0x00 sav_pri:0x7f

Misc:  lockcount:0x00000000 ulock:0x00000000 *graphics:0x00000000
  dispct:0x006ecdcf fpuct:0x00000000 *cancel:0x00000000 boosted:0x0000
Signal Information:  cursig:0x00 *scp:0x00000001
  pending:hi 0x00000000,lo 0x00000000 sigmask:hi 0x00000000,lo 0x00000000


SLT ST   TID    PID   CPUID  POLICY PRI CPU   EVENT  PROCNAME    FLAGS
 3 s    307    306      0     RR  24  0           netm
     t_flags:  sigslih kthread

Links:  *procp:0xe3000300 *uthreadp:0x2ff3b400 *userp:0x2ff3b6e0
  *prevthread:0xe6000240 *nextthread:0xe6000240,  *stackp:0x00000000
  *wchan1(real):0x00000000  *wchan2(VMM):0x00000000 *swchan:0x00000000
  wchan1sid:0x00000000  wchan1offset:0x00000000
  pevent:0x00000000  wevent:0x00000912  *slist:0x00000000
Dispatch Fields:  *prior:0xe6000240  *next:0xe6000240
  polevel:0x00000000  ticks:0x0001  *synch:0xffffffff  result:0x00000000
  *eventlst:0x00000000  *wchan(hashed):0x00000000  suspend:0x0001
  thread waiting for:  event(s)
```



```
Scheduler Fields:  cpuid:0x0000  scpuid:0x0000  pri: 36  policy:RR
  affinity:0x0000 cpu:0x0000   lpri:  0 wpri:127   time:0xff sav_pri:0x24

Misc:  lockcount:0x00000000 ulock:0x00000000 *graphics:0x00000000
  dispct:0x00008d3f fpuct:0x00000000 *cancel:0x00000000 boosted:0x0000
Signal Information:  cursig:0x00 *scp:0x00000001
  pending:hi 0x00000000,lo 0x00000000 sigmask:hi 0x00000000,lo 0x00000000
```

Lab Exercise 2

Use the crash command to examine the state of the currently running process. Use the "u" command to list the user block.

You should see output similar to this:

```
> u
      UTHREAD AREA FOR SLOT  29

SAVED MACHINE STATE
   curid:0x00001cd4  m/q:0x00000000  iar:0x10011f60  cr:0x28826828
   msr:0x4000f0b0  lr:0x100015e0  ctr:0x100015bc  xer:0x00000002
   *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0000000b
   backtrace:0x00  tid:0x00000000  fpeu:0x01  ecr:0x00000000
   Exception Struct
    0x10011f60  0x4000f0b0  0x40000343  0x10011f60  0x00000107
   Segment Regs
    0:0x40000707   1:0x40000343   2:0x40000e8e   3:0x007fffff
    4:0x007fffff   5:0x007fffff   6:0x007fffff   7:0x007fffff
    8:0x007fffff   9:0x007fffff  10:0x007fffff  11:0x007fffff
```



```
   12:0x007fffff  13:0x40000a0a  14:0x007fffff  15:0x40001494
   General Purpose Regs
    0:0x00000001   1:0x2ff20f70   2:0x20014598   3:0xffffffff
    4:0x00000000   5:0x2002c18a   6:0x200019e0   7:0x00007500
    8:0x00000000   9:0x40000343  10:0x00000001  11:0x000034cc
   12:0x100015c4  13:0x20001730  14:0x20001ea8  15:0x20001ee8
   16:0xf02995a0  17:0x00000000  18:0x200149e0  19:0x200149d8
   20:0x200017b4  21:0x200016a4  22:0x20001724  23:0x20001708
   24:0x20001720  25:0x2001bc88  26:0x20001718  27:0x2000170c
   28:0x20001784  29:0x2000209c  30:0x00000000  31:0x00000000
   Floating Point Regs
      Fpscr: 0x00000000
    0:0x00000000 0x00000000  1:0x00000000 0x00000000  2:0x00000000 0x00000000
    3:0x00000000 0x00000000  4:0x00000000 0x00000000  5:0x00000000 0x00000000
    6:0x00000000 0x00000000  7:0x00000000 0x00000000  8:0x00000000 0x00000000
    9:0x00000000 0x00000000 10:0x00000000 0x00000000 11:0x00000000 0x00000000
   12:0x00000000 0x00000000 13:0x00000000 0x00000000 14:0x00000000 0x00000000
   15:0x00000000 0x00000000 16:0x00000000 0x00000000 17:0x00000000 0x00000000
   18:0x00000000 0x00000000 19:0x00000000 0x00000000 20:0x00000000 0x00000000
   21:0x00000000 0x00000000 22:0x00000000 0x00000000 23:0x00000000 0x00000000
   24:0x00000000 0x00000000 25:0x00000000 0x00000000 26:0x00000000 0x00000000
```

```
   27:0x00000000 0x00000000 28:0x00000000 0x00000000 29:0x00000000 0x00000000
   30:0x00000000 0x00000000 31:0x00000000 0x00000000

Kernel stack address: 0x2ff3b400

SYSTEM CALL STATE
   error code:0x00  *kjmpbuf:0x00000000
      ut_flags:  assertsig

PER-THREAD TIMER MANAGEMENT
   Real/Alarm Timer (ut_timer[TIMERID_ALRM]) = 0x0
   Virtual Timer (ut_timer[TIMERID_VIRTUAL]) = 0x0
   Prof Timer (ut_timer[TIMERID_PROF]) = 0x0
   Posix Timer (ut_timer[POSIX0]) = 0x0
   Posix Timer (ut_timer[POSIX1]) = 0x0
   Posix Timer (ut_timer[POSIX2]) = 0x0
   Posix Timer (ut_timer[POSIX3]) = 0x0
   Posix Timer (ut_timer[POSIX4]) = 0x0

SIGNAL MANAGEMENT
   *sigsp:0x0  oldmask:hi 0x0,lo 0x0  code:0x0
```



```
MISCELLANOUS FIELDS:
  fstid:0x00000000  ioctlrv:0x00000000  selchn:0x00000000
  link:0x00000000  loginfo:0x00000000  fselchn:0x00000000
  selbuc:0x00000000  sigssz:0x00000000


     USER AREA OF ASSOCIATED PROCESS crash (SLOT  28, PROCTAB 0xe3001c00)
   handy_lock:0x00000000  timer_lock:0x00000000
   map:0x00000000  *semundo:0x00000000
   compatibility:0x00000000  lock:0x00000000
   ulocks:0xffffffff
SIGNAL MANAGEMENT
   Signals to be blocked (sig#:hi/lo mask,flags,&func)
    1:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
    2:hi 0x00000000,lo 0x00000000,0x00000002,0x20014490
    3:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
    4:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
    5:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
    6:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
    7:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
```

```
 8:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
 9:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
10:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
11:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
12:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
13:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
14:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
15:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
16:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
17:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
18:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
19:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
20:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
21:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
22:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
23:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
24:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
25:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
26:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
27:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
28:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
```



```
29:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
30:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
31:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
32:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
33:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
34:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
35:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
36:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
37:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
38:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
39:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
40:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
41:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
42:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
43:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
44:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
45:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
46:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
47:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
48:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
49:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
```

```
  50:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  51:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  52:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  53:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  54:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  55:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  56:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  57:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  58:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  59:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  60:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  61:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  62:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
  63:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000


USER INFORMATION
  euid:0x0000 egid:0x0000 ruid:0x0000 rgid:0x0000 luid:0x00000000
  suid:0x00000000 ngrps:0x0006 *groups:0x2ff20e48 compat:0x00000000
  ref:0x00000005
  acctid:0x00000000  sgid:0x00000000  epriv:0xffffffff
```



```
  ipriv:0xffffffff  bpriv:0xffffffff  mpriv:0xffffffff

  u_info: LOGNAME=root

ACCOUNTING DATA
  start:0x31af87e8 ticks:0x002198cd acflag:0x0002 pr_base:0x00000000
  pr_size:0x00000000 pr_off:0x00000000 pr_scale:0x00000000
  process times:
       user:0x00000002s 0x01c9c380us
        sys:0x00000003s 0x07270e00us
  children's times:
       user:0x00000000s 0x00000000us
        sys:0x00000000s 0x00000000us

CONTROLLING TTY
  *ttysid:0x05a9a10c *ttyp(pgrp):0x05a9a108
  ttyd(evice):0x00190000 ttympx:0x00000000 *ttys(tate):0x00000000
  tty id: 0x05a9a108 *query function: 0x018fbdcc
PINNED PROFILING BUFFER
  *pprof: 0x00000000 *mem desc: 0x00000000
```

```
RESOURCE LIMITS AND COUNTERS
  ior:0x00000000  iow:0x00000000  ioch:0x01525dbc
  text:0x00022c30  data:0x0020d000  stk:0x02000000
  max data:0x08000000  max stk:0x02000000  max file:0x3ffffe00
  soft core dump:0x00100000  hard core dump:0x7fffffff
  soft rss:0x02000000  hard rss:0x7fffffff
  cpu soft:0x7fffffff  cpu hard:0x7fffffff
  hard ulimit:0x3ffffe00
  minflt:0x00000000   majflt:0x00000000

AUDITING INFORMATION
  auditstatus:0x00000000

SEGMENT REGISTER INFORMATION
  Reg      Flag     Fileno    Pointer
  0        0        0         0
  1        0        0         0
  2        0        0         0
  3        0        0         0
  4        0        0         0
  5        0        0         0
```



```
  6        0        0         0
  7        0        0         0
  8        0        0         0
  9        0        0         0
  10       0        0         0
  11       0        0         0
  12       0        0         0
  13       80       0         0
  14       0        0         0
  15       20       0         0
  *adspace:0xa0050000
FILE SYSTEM STATE
  *curdir:0x0626ae28  *rootdir:0x00000000
  cmask:0x0012  maxindex:0x0007
FILE DESCRIPTOR TABLE
  *ufd: 0x20015754
  fd 0:  fp = 0x0027b7e8   flags = 0x00000080
  fd 1:  fp = 0x0027b7e8   flags = 0x00000080
  fd 2:  fp = 0x0027b7e8   flags = 0x00000080
  fd 3:  fp = 0x0027b788   flags = 0x00000080
  fd 4:  fp = 0x0027bf98   flags = 0x00000080
```

```
  fd 5:  fp = 0x0027bd28   flags = 0x00000080
  fd 6:  fp = 0x0027bf08   flags = 0x00000081
> mst
          MSTSAVE AREA AT ADDRESS 0x2ff3b400

  curid:0x00001cd4  m/q:0x00000000  iar:0x10017a34  cr:0x28826828
  msr:0x4000f0b0   lr:0x10002a84  xer:0x00000004  kjmpbuf:0x00000000
  backtrack:0x00   tid:0x00000000  fpeu:0x01      excp_type:0x00000000
  ctr:0x10002a70
  *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0b
  o_iar:0x00000000  o_toc:0x00000000  o_arg1:0x00000000  excbranch:0x0007fe68
  o_vaddr:0x00000000
  Exception Struct
   0x10017a34  0x4000f0b0  0x40000343  0x10017a34  0x00000107
  Segment Regs
   0:0x40000707   1:0x40000343   2:0x40000e8e   3:0x007fffff
   4:0x007fffff   5:0x007fffff   6:0x007fffff   7:0x007fffff
   8:0x007fffff   9:0x007fffff  10:0x007fffff  11:0x007fffff
  12:0x007fffff  13:0x40000a0a  14:0x007fffff  15:0x40001494
  General Purpose Regs
   0:0x00000001   1:0x2ff20f70   2:0x20014598   3:0x00000000
```



```
   4:0x200019d4   5:0x2002c18c   6:0x200019e0   7:0x6d737400
   8:0x00000000   9:0x40000343  10:0x00000001  11:0x000034cc
  12:0x10002a74  13:0x20001730  14:0x20001ea8  15:0x20001ee8
  16:0xf02995a0  17:0x00000000  18:0x200149e0  19:0x200149d8
  20:0x200017b4  21:0x200016a4  22:0x20001724  23:0x20001708
  24:0x20001720  25:0x2001bc88  26:0x20001718  27:0x2000170c
  28:0x20001784  29:0x2000209c  30:0x2000ead0  31:0x2000ead0
```

Lab Exercise 3

Use the crash command to examine the file system data structures. Use the "file" command to list the open file table.

You should see output similar to this:

```
> file
SLOT   FILE       REF    TYPE          ADDRESS     FLAGS
  0  0x0027a018   19   inode.VREG       0x06270a98  0x00001001 read
  1  0x0027a048    1   inode.VREG       0x06266178  0x00000020
  2  0x0027a078    6   inode.VREG       0x0626b898  0x00001001 read
  3  0x0027a0a8    5   inode.VREG       0x0627a5a0  0x00001001 read
  4  0x0027a0d8    2   inode.VREG       0x0628ad90  0x00001001 read
  5  0x0027a108    1   inode.VREG       0x062838f8  0x00000003 read write
  6  0x0027a138    1   inode.VCHR       0x063c71a8  0x00000001 read
  7  0x0027a168    1   inode.VREG       0x06282850  0x00000020
  8  0x0027a198    2   inode.VREG       0x06271620  0x00001001 read
  9  0x0027a1c8    2   inode.VCHR       0x063c7140  0x04000002 write
 10  0x0027a1f8    1   inode.VCHR       0x063c70d8  0x00000001 read
 11  0x0027a228    3   inode.VREG       0x06279640  0x00001001 read
```



```
 12  0x0027a258    1   inode.VREG       0x06275258  0x00000020
 13  0x0027a288    1   inode.VCHR       0x063c78d8  0x00000002 write
 14  0x0027a2b8    2   inode.VREG       0x0626fb38  0x00001001 read
 15  0x0027a2e8    1   inode.VCHR       0x063c7140  0x00000001 read
 16  0x0027a318    1   inode.VDIR      0x06270950  0x00000001 read
 17  0x0027a348    1   inode.VREG       0x06282478  0x00000003 read write
 18  0x0027a378    1   inode.VCHR       0x063c7940  0x00000002 write
 19  0x0027a3a8    5   inode.VREG       0x062662c0  0x00001001 read
 20  0x0027a3d8    1   inode.VCHR       0x063c7348  0x20008802 write ndelay
```

Use the "vfs" command to examine the vfs list.

You should see output similar to this:

```
> vfs
VFS ADDRESS   TYPE OBJECT    STUB NUMBER FLAGS  PATHS
 0 5494838    jfs 626ae28       0     5 D     /dev/hd4 mounted over /
 1 5494898    jfs 6270e40 626f730     6 D     /dev/hd2 mounted over /usr
 2 5494868    jfs 6271738 6271360     7 D     /dev/hd9var mounted over /var
 3 5494808    jfs 62687b8 6268298     8 D     /dev/hd3 mounted over /tmp
 4 54948c8    jfs 628bf50 6285120     9 D     /dev/hd1 mounted over /home

      flags:  C=disconnected D=device   I=remote P=removable
           R=readonly     S=shutdown U=unmounted Y=dummy
```


Use the "vfs -" command to examine the vfs list and the vnodes on each vfs.

You should see output similar to this:

```
> vfs -
VFS ADDRESS   TYPE OBJECT    STUB NUMBER FLAGS  PATHS
 0 5494838    jfs 626ae28       0     5 D     /dev/hd4 mounted over /
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT    INODE FLAGS
628c5b8  0   -  vreg   jfs    0    -  628c5e8
6285788  0   -  vreg   jfs    0    -  62857b8
6287128  0   -  vreg   jfs    0    -  6287158
628c848  0   -  vreg   jfs    0    -  628c878
626a158  0   -  vreg   jfs    0    -  626a188
62853b0  0   -  vreg   jfs    0    -  62853e0
6283f30  0   -  vreg   jfs    0    -  6283f60
627ad20  0   -  vreg   jfs    0    -  627ad50
62858d0  0   -  vreg   jfs    0    -  6285900
627b8a8  0   -  vreg   jfs    0    -  627b8d8
6273c60  0   -  vreg   jfs    0    -  6273c90
627a050  0   -  vlnk   jfs    0    -  627a080
62739d0  0   -  vlnk   jfs    0    -  6273a00
```

```
 1 5494898    jfs 6270e40 626f730     6 D    /dev/hd2 mounted over /usr
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT    INODE FLAGS
6278048  1   -  vreg   jfs    2    - 6278078
6286458  1   -  vreg   jfs    0    - 6286488
62866e8  1   -  vreg   jfs    0    - 6286718
627d4d8  1   -  vreg   jfs    0    - 627d508
6275228  1   -  vlnk   jfs    0    - 6275258
626e3f8  1   -  vreg   jfs    0    - 626e428
626a678  1   -  vlnk   jfs    0    - 626a6a8
627b760  1   -  vreg   jfs    0    - 627b790
6273368  1   -  vreg   jfs    0    - 6273398
6278bd0  1   -  vreg   jfs    0    - 6278c00
628f520  1   -  vreg   jfs    2    - 628f550
6267ae8  1   -  vreg   jfs    1    - 6267b18
62826d8  1   -  vreg   jfs    0    - 6282708
628b7a0  1   -  vreg   jfs    0    - 628b7d0

 2 5494868    jfs 6271738 6271360     7 D    /dev/hd9var mounted over /var
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT    INODE FLAGS
6266cd0  2   -  vdir   jfs    0    - 6266d00
```



```
6269ec8  2   -  vdir   jfs    0    - 6269ef8
628fe18  2   -  vdir   jfs    0    - 628fe48
62767f0  2   -  vreg   jfs    0    - 6276820
62770e8  2   -  vdir   jfs    0    - 6277118
6274a78  2   -  vdir   jfs    1    - 6274aa8
6274038  2   -  vreg   jfs    0    - 6274068
626e7d0  2   -  vdir   jfs    1    - 626e800
6286e98  2   -  vreg   jfs    1    - 6286ec8
628d3d0  2   -  vdir   jfs    1    - 628d400
628ceb0  2   -  vfifo  jfs    1    - 628cee0
626b200  2   -  vdir   jfs    1    - 626b230
6282590  2   -  vreg   jfs    1    - 62825c0
6282448  2   -  vreg   jfs    2    - 6282478
6281f28  2   -  vdir   jfs    1    - 6281f58
6271738  2   -  vdir   jfs    1    - 6271768 vfs_root

 3 5494808    jfs 62687b8 6268298     8 D    /dev/hd3 mounted over /tmp
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT    INODE FLAGS
62687b8  3   -  vdir   jfs    2    - 62687e8 vfs_root

 4 54948c8    jfs 628bf50 6285120     9 D    /dev/hd1 mounted over /home
```

```
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT   INODE FLAGS
628bf50  4   -  vdir   jfs    1     -  628bf80 vfs_root
```


Use the "inode" command to examine the inode table.

You should see output similar to this:

```
> inode
 ADDRESS   MAJ  MIN INUMB REF LINK  UID  GID   SIZE   MODE  SMAJ SMIN FLAGS
0x0626a6a8 010 0005   413   0    1    2    2     0 ----777   -    -
0x06270a98 010 0005   409   1    1    2    2     0 f---755   -    -
0x06269ef8 010 0004   433   0    1    0    0     0 c---666 0031 0167
0x06266d00 010 0004   434   0    1    0    0     0 c---666 0031 0166
0x06283f60 010 0005  9110   0    1    2    2     0 f---644   -    -
0x0626e428 010 0004   435   0    1    0    0     0 c---666 0031 0165
0x06275258 010 0005   445   1    2    2    2     0 f---555   -    -
0x06286718 010 0004   444   0    1    0    0     0 c---666 0031 0154
0x06285900 010 0004   445   0    1    0    0     0 c---666 0031 0153
0x0627ad50 010 0005  8605   0    1    2    2     0 f---644   -    -
0x062853e0 010 0005  9107   0    1    2    2     0 f---644   -    -
0x0626a188 010 0004   446   0    1    0    0     0 c---666 0031 0152
0x0627d508 010 0004   447   0    1    0    0     0 c---666 0031 0151
0x0628c878 010 0004   440   0    1    0    0     0 c---666 0031 0160
0x06282c28 010 0005  8607   1    1    2    2     0 f---644   -    -
```

```
0x06287158 010 0004   441  0   1   0   0    0 c---666 0031 0157
0x062857b8 010 0004   442  0   1   0   0    0 c---666 0031 0156
```


Lab Exercise 4

Use the crash command to examine the state of the currently running thread. Use the "mst" command to look at the machine state save area(s) for the currently running process.

You should see output similar to this:

```
> mst
          MSTSAVE AREA AT ADDRESS 0x2ff3b400

  curid:0x00001864  m/q:0x00000000  iar:0x10017a34  cr:0x28826428
  msr:0x4000d0b0   lr:0x10002a84   xer:0x00000004  kjmpbuf:0x00000000
  backtrack:0x00   tid:0x00000000  fpeu:0x01      excp_type:0x00000000
  ctr:0x10002a70
  *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0b
  o_iar:0x00000000  o_toc:0x00000000  o_arg1:0x00000000  excbranch:0x0007fe68
  o_vaddr:0x00000000
  Exception Struct
   0x10017a34  0x4000d0b0  0x40000747  0x10017a34  0x00000107
  Segment Regs
   0:0x40000707   1:0x40000747   2:0x40001637   3:0x007fffff
```

```
 4:0x007fffff  5:0x007fffff  6:0x007fffff  7:0x007fffff
 8:0x007fffff  9:0x007fffff 10:0x007fffff 11:0x007fffff
12:0x007fffff 13:0x40000a0a 14:0x007fffff 15:0x40000726
General Purpose Regs
 0:0x00000001  1:0x2ff20f70  2:0x20014598  3:0x00000000
 4:0x200019d4  5:0x2002c18c  6:0x200019e0  7:0x6d737400
 8:0x00000000  9:0x40000747 10:0x00000001 11:0x000034cc
12:0x10002a74 13:0x20001730 14:0x20001ea8 15:0x20001ee8
16:0xf02995a0 17:0x00000000 18:0x200149e0 19:0x200149d8
20:0x200017b4 21:0x200016a4 22:0x20001724 23:0x20001708
24:0x20001720 25:0x2001bc88 26:0x20001718 27:0x2000170c
28:0x20001784 29:0x2000209c 30:0x2000ead0 31:0x2000ead0
```


Use the "trace" command to see the current call path of the current process.

You should see output similar to this:

```
> trace
STACK TRACE:
      .vnop_rdwr ()
      .rwuio ()
      .rdwr ()
      .kreadv ()
      .sys_call ()
```

Use the "trace -m" command to see the current call path and mstsave areas of the current process.

You should see output similar to this:

```
> trace -m
MST STACK TRACE:
   0x2ff3b400 (excpt=3021e000:40000000:1dfd:3021e000:106) (intpri=b)
      IAR:      10006018 <invalid>:   rlimi.  r11,r15,0x10,20,15
      *LR:      100012ec <invalid>
      *2ff22d50: 10001514 <invalid>
IAR not in kernel segment.
```


Use the "trace -k" command to see the current call path of the current process with more information.

You should see output similar to this:

```
> trace -k
STACK TRACE:
   0x1cd4 (excpt=20004b4c:40000000:40000e8e:20004b4c:106) (intpri=b)
      IAR:      018e3104 [pse:sth_rput] + 12ea4:    cmpi   cr0,r31,0x0
      *LR:      018e29f0 [pse:sth_rput] + 12790
      *2ff3aa48: 018e29f0 [pse:sth_rput] + 12790
      2ff3aaa8:  018d977c [pse:csq_run] + 951c
      2ff3ab08:  018d88e4 [pse:csq_lateral] + 8684
      2ff3ab58:  018d5630 [pse:putnext] + 53d0
      2ff3abb8:  019aeac8 [ptydd:ptm_rput] + 4188
      2ff3ac18:  018d977c [pse:csq_run] + 951c
      2ff3ac78:  018d88e4 [pse:csq_lateral] + 8684
      2ff3acc8:  018d5784 [pse:puthere] + 5524
      2ff3ad18:  019aba4c [ptydd:ptm_put] + 110c
      2ff3ad68:  019ab8e4 [ptydd:pty_ptm_xfer] + fa4
      2ff3adc8:  019b1b4c [ptydd:pts_wput] + 720c
      2ff3ae18:  018d5574 [pse:putnext] + 5314
```

```
      2ff3ae78:  01984488 [ldterm:ldtty_canputnext] + 46e8
      2ff3aee8:  019849f0 [ldterm:ldtty_write] + 4c50
      2ff3af58:  0198b228 [ldterm:ldtty_wput] + b488
      2ff3af98:  018d3bdc [pse:str_putnext] + 397c
      2ff3b008:  018f09fc [pse:osr_write] + 2079c
      2ff3b068:  018da47c [pse:osr_run] + a21c
IAR not in kernel segment.
```


Use the "fs" command to see a formatted stack dump of the current process.

You should see output similar to this:

```
> fs
STACK TRACE:

      **** .e_block_thread ****
2ff3aee8  2FF3AF58 48224825 0198B228 05AA6708   /..XH"H%...(..g.
2ff3aef8  2FF3AF48 01991CB8 05AA668C 05AA6708   /..H.....f...g.
2ff3af08  2FF3AF48 20224022 018DA898 0000000B   /..H "@".....
2ff3af18  05AAAF00 018FC978 0000000B 00000000   .....x.....
2ff3af28  018FAAD0 05AA6A00 00000015 00000000   .....j.....
2ff3af38  00000001 05AA688C 00000000 00004000   .....h.....@.
2ff3af48  00000000 05AA6844 05AA7600 05AA648C   ..(...hD..v...d.

      **** 018d3bdc ****
2ff3af58  2FF3AF98 44224824 018D3BDC 00000000   /...D"H$..;.....
2ff3af68  2FF3C110 018FC978 05A9A10C 05A9A108   /.....x.....
2ff3af78  00000001 05AA688C 00000000 00004000   .....h.....@.
2ff3af88  00000000 05AA6844 05A8E400 05AA685C   .....hD.....h\
```

```
      **** 018f09fc ****
2ff3af98  2FF3B008 44225824 018F09FC 018FA940   /...D"X$.....@
2ff3afa8  2FF3B008 0007FB44 018D8B78 00008000   /......D...x....
2ff3afb8  2FF3B038 018FC978 00000020 2FF3B245   /..8...x... /..E
2ff3afc8  2FF3B008 018FC978 00000000 2002C398   /......x.... ...
2ff3afd8  D00AEC30 FFFFFFF8 D00AE358 2FF20D40   ...0.....X/..@
2ff3afe8  2FF3B6E0 05AA6A50 0000007B 05AA6A48   /.....jP...{..jH
2ff3aff8  00000000 05AA6A00 05AA6AA0 05AA685C   ......j...j...h\

      **** 018da47c ****
2ff3b008  2FF3B068 00000000 018DA47C 007FFFFF   /..h.......|....
2ff3b018  2FF3B068 018FC978 0628F550 0007FFFF   /..h...x.(.P....
2ff3b028  2FF3B078 2FF3B338 00000224 00001DFD   /..x/..8...$....
2ff3b038  2FF3B0B8 018FC978 00030194 00000000   /......x........
2ff3b048  2FF3B6E0 2FF3B400 31AF897F 0CA42A00   /.../...1.....*.
2ff3b058  00000000 062667E0 00000060 2FF3B150   .....&g....`/..P
IAR not in kernel segment.
```


Use the "stack" command to see an unformatted stack dump of the current process.

You should see output similar to this:

```
> stack
KERNEL STACK:

2ff3aee0:       197fea0  5aa648c  2ff3af58  48224825
2ff3aef0:       198b228  5aa6508  2ff3af48  1991cb8
2ff3af00:       5aa648c  5aa6508  2ff3af48  20224022
2ff3af10:       18da898        b  5aaaf00  18fc978
2ff3af20:             b        0  18faad0  5aa6a00
2ff3af30:             e        0        1  5aa688c
2ff3af40:             0     4000        0  5aa6844
2ff3af50:       5a8e100  5aa648c  2ff3af98  44224824
2ff3af60:       18d3bdc        0  2ff3c110  18fc978
2ff3af70:       5a9a10c  5a9a108        1  5aa688c
2ff3af80:             0     4000        0  5aa6844
2ff3af90:       5a8e100  5aa685c  2ff3b008  44225824
2ff3afa0:       18f09fc  18fa940  2ff3b008     7fb44
2ff3afb0:       18d8b78     8000  2ff3b038  18fc978
```

```
2ff3afc0:          20  2ff3b245  2ff3b008   18fc978
2ff3afd0:           0  2002c390  d00aec30         0
2ff3afe0:    d00ae358  2ff2054c  2ff3b6e0   5aa6a50
```


Lab Exercise 5

Use the crash command to examine the current usage of the kernel and pinned heaps and the loader data structures.

Use the "xmalloc" command to see the current usage of the kernel and pinned heaps.

You should see output similar to this:

```
> xmalloc
Kernel heap usage
heap size = 242454528 amount used = 79659872
Pinned heap usage
heap size = 242454528 amount used =  274512
Kernel and pinned heap usage
```

Use the "le" command to see the current loader entries for objects loaded in the kernel domain.

You should see output similar to this:

```
> le
LoadList entry at 0x05532b00
  Module start:0x019e5260  Module filesize:0x00002d90  *end:0x019e7ff0
  *data:0x019e7cc0  data length:0x00000330
  Use-count:0x0003  load_count:0x0001  *file:0x00000000
  flags:0x00000272 TEXT EXTN DATX DATA DAEX
  *exp:0x064b3000  *lex:0x00000000  *deferred:0x00000000
  Name: /etc/drivers/tok_demux
  ndepend:0x0001  maxdepend:0x0001
  *depend[00]:0x05532780
  le_next:  05532000

LoadList entry at 0x05532000
  Module start:0x064be000  Module filesize:0x00000c30  *end:0x064bec30
  *data:0x00000000  data length:0x00000000
  Use-count:0x0006  load_count:0x0000  *file:0x00000000
  flags:0x01000248 SYSC DATA DAEX
```



```
  *exp:0x064bf000  *lex:0x00000000  *deferred:0x00000000
  Name: /unix
  ndepend:0x0002  maxdepend:0x0002
  *depend[00]:0x05532a80
  *depend[01]:0x05532b00
  le_next:  05532b80

LoadList entry at 0x05532b80
  Module start:0x0195aea0  Module filesize:0x00012038  *end:0x0196ced8
  *data:0x0196b080  data length:0x00001e58
  Use-count:0x0001  load_count:0x0001  *file:0x00000000
  flags:0x00000262 TEXT DATX DATA DAEX
  *exp:0x00000000  *lex:0x00000000  *deferred:0x00000000
  Name: /etc/drivers/tokdd
  ndepend:0x0001  maxdepend:0x0001
  *depend[00]:0x05532780
  le_next:  05532700

LoadList entry at 0x05532700
  Module start:0x019e24c0  Module filesize:0x00002d90  *end:0x019e5250
  *data:0x019e4f20  data length:0x00000330
```

```
Use-count:0x0001  load_count:0x0000  *file:0x00000000
flags:0x00000273 UNLD TEXT EXTN DATX DATA DAEX
*exp:0x06499000  *lex:0x00000000  *deferred:0x00000000
Name: /etc/drivers/tok_demux
ndepend:0x0001  maxdepend:0x0001
*depend[00]:0x05532780
le_next:  05532c00
```