Accelerating AIX

Performance Tuning for Programmers and System Administrators

Rudy Chukran

The publisher offers discounts of this book when ordered in quantity for special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

Library of Congress Cataloging-in-Publication Data

Chukran, Rudy, 1952–
Accelerating AIX : performance tuning for programmers and system administrators / Rudy Chukran.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-63382-5
1. AIX (Computer file) 2. Operating systems (Computers)
I. Title.
QA76.76.063C495 1998
005.4'469—dc21 97–46800
CIP

Text printed on recycled paper
ISBN 0-201-63382-5
4 5 6 7 8 9—CRE—07 06 05 04 03
Fourth Printing, April 2005

2

AIX System Design

In this chapter you will learn basic design points about the AIX that affect system performance. The topics you will study are virtual memory, the program loader, the filesystem, process scheduler, and the network subsystem. After reading this chapter, you will understand why simply adding real memory can drastically speed up both program execution and program I/O efficiency. You will understand why programs linked with shared libraries can save lots of virtual memory in some cases yet not save any virtual memory in others. You will also learn how processes are chosen for dispatching and why your favorite application processes may not be so favored as far as the system scheduler is concerned.

Before you can begin to adapt and optimize a computer system to perform at its fullest advantage, it is helpful to learn how the system is put together. You don't need to know minute details as well as the developer who wrote the code, but the more you know, the more effective you will be at tuning your system. I will use an analogy that compares cars to computers.

The computer system user is like a passenger in an automobile. It requires very little understanding of how the car is designed in order to be a passenger. If you know how the door works and how the seat belt buckles, you have the passenger skills licked. If you want to be the chauffeur or a system administrator, you need more design knowledge. The chauffeur needs to understand how to steer and accelerate. Some basic engine and braking design must be understood so that the brake and accelerator are not depressed at the same time. Likewise, the system administrator must not attempt to configure the system to perform conflicting tasks. However, if you want to be a mechanic to fix problems with a sputtering engine, you will need a much more complete understanding of the design of the car. The better you understand the overall engine design, the more effective mechanic you will be. And the better you understand the design of AIX, the better you will be at being an AIX performance mechanic.

First, I'll assume that you already have a basic background on how a generic computer system works. That is, you have already taken Introduction to Computers 101, and you already know what binary arithmetic is about. I will also assume you are familiar with UNIX in general and know some of the basic design concepts. For example, you should know what virtual memory is and why it is useful, what UNIX filesystems are, and what the basic commands are that UNIX users must know to manipulate the system.

In order to understand the finer points of the AIX design, we will describe some scenarios that you know and show a little bit of what is going on behind the scenes. To do this, we will take a tour of the guts of the AIX operating system when you execute a command. We will see the interesting things that happen until the program you want to run receives control at its first statement in the main subroutine. Once our sample program gets control and begins executing, we will follow its execution path, that is, the system subroutines.

2.1 General AIX Design Goals

The AIX operating system kernel was one of the first UNIX kernels to be designed with modularity and extendibility in mind. Several design factors contribute to these goals:

Loadable device drivers

Pageable kernels

Dynamically growing kernel tables

Device driver support is not built into the core kernel. Instead, kernel device drivers are dynamically loaded and unloaded at runtime. This technique contrasts with the old way of needing to relink the kernel whenever new device support is required. AIX handles this by packaging the device driver with installation methods that load the driver and all the configuration data. This means that adding a new device driver for an existing device does not require a reboot.

The AIX kernel is mostly pageable. Except for device interrupt handlers that must be pinned into memory, most of the kernel is pageable. This allows the kernel to take a much smaller real memory footprint than it would if it were not pageable.

As a result of the kernel being pageable, kernel tables are able to grow on demand. Tables such as the process table are memory mapped; then entire large tables are assigned a virtual address range. The virtual entries are assigned to real memory as they are referenced. When a new entry of the process table is referenced, a page fault occurs, and a new virtual page is allocated. All of the kernel tables are allocated this way. They are mapped into virtual memory with a very large size and paged in as needed. For example, the process table is allocated with 131,071 (2^{17}-1) entries. Neither virtual memory nor real memory is used until the entry is really needed. The end result of this dynamic table allocation is that the AIX kernel does not need to be recompiled to make tables larger as is the case with other UNIX kernels.

2.2 Program Loading

The first order of business is to examine how programs are loaded in a virtual memory space and how control is given to the first executable statement. You start the whole process by typing a command at the shell prompt. The shell forks a child process, which then passes the string as an argument to the exec system call. The exec system call switches to the kernel system call handler that ends up calling the kernel loader. The loader is responsible for loading programs into virtual memory and resolving the various symbolic references into virtual addresses. This symbol resolution is done at the time the program is being loaded and will be finished by the time the exec system call is complete. This concept of resolving symbols at the time of the exec system call (which I call exec time for short) is called *dynamic loading*.

Dynamic loading is important because it enables you to write programs that can use less memory as compared with programs that do not use this facility. (We will get into more detail on symbol resolution in Chapter 6 when we talk about writing programs to take advantage of dynamic symbol resolution.) The loader locates the instruction portion of the executable and maps it to the text segment. The act of mapping is just a way of associating a part of a disk file to a portion of virtual memory; no copying of the executable text portion is done.

The text portion is not read into storage by the loader. This feature is useful for very large programs because the time is not wasted waiting for the whole program to be read in. As the pages in the address range are referenced, the corresponding page of the executable file is paged in. This technique is referred to as *demand paging*. If sections of code are never executed, then the time to read this code into virtual memory is not wasted, nor is the real memory used to hold this code wasted.

The most obvious mapping assignment that the loader needs to make is the text mapping. For AIX, text is mapped into a hardware segment. AIX implementations on Power and PowerPC rely on the fact that these architectures are segmented in which the 32-bit virtual address space is divided into 16 segments of 256 megabytes each. Each of these segments can be assigned its own protection domain. Text is mapped into its own read-only segment.

However, there are other segment assignments that the loader needs to make. See Figure 2.1. Segment 0 is always mapped to the kernel instructions and data. While the process is running in the user mode, as opposed to the kernel mode, the kernel segment is assigned read-only protection. User code actually requires the ability to read the kernel because user programs can execute code in the kernel while in user mode. Segment 1 is assigned the process main text segment. This is the part of a program that is executed and contains a main statement. Segment 2 is assigned to the private data segment. This segment is subdivided into program initialized data, program uninitialized data, program stack, and kernel stack. Segment 13 is assigned to shared library text; segment 15 is assigned to shared library data. This leaves several segments unassigned. The unassigned segments, 3 through 12, are available for use as auxiliary data segments, shared data segments between processes, or segments in which to map files.

At this point, the loader has mapped text and data into the proper portions of the executable file. The symbol resolution occurs after the text and data segments have been

Segment	Start Address		End Address
0	00000000	Kernel Segment	0 FFFFFFF
1	10000000	Program Text Segment	1 FFFFFFF
2	20000000	Program Data/Stack Segment	2 FFFFFFF
3-C	30000000	Available as Shared Memory Segments or Extended Program Data Segments	CFFFFFF
D	D0000000	Shared Library Text Segment	DFFFFFFF
E	E0000000		EFFFFFFF
F	F0000000	Shared Library Data Segment	FFFFFFFF

Figure 2.1 Hardware segment assignments

mapped. Symbol resolution assigns to each symbol in a program an address in the virtual address space of the process that is about to execute. The loader reads a portion of the executable called the loader section. This portion is a list of all of the symbols that must be assigned addresses. Once this address assignment is complete, the program is ready to run. All that is necessary is to initialize the program counter to point to the first instruction in the program. Program counter initialization is accomplished in quite a sneaky way. When the exec system call returns, it returns to the new program at the first instruction rather than return to the point of invocation, as most other system calls do. The loader's job is done once the new program counter is initialized, and this signifies completion of the exec system call.

It is interesting to review which parts of the executable have actually been copied from the image on disk to the image in memory. These are only the executable header and the symbol dictionary. The header is a kind of road map that tells the loader where the various parts of the executable are located, and the symbol dictionary maps symbols into addresses. Once these two parts have been read and their use is complete, they are discarded. The actual code that is about to run has not been touched at all.

2.3 Virtual Memory Overview

When the first address is first touched, a *page fault* occurs. Page faults drive the virtual memory system to perform on demand. It is probably a good time to examine the virtual memory manager (VMM) at this point, because that will prepare us for the read and write system calls that most programs perform. A simple way to describe the effect of the virtual memory manager is to say that as the data is needed, it is fetched on demand. This concept is called *demand paging*.

2.3.1 Page Faults

Let's look at this concept of fetching data from disk on demand, the basis of how virtual memory is managed on AIX. First, let's review some terminology. Data is organized in units of pages that are 4096 bytes in size for the Power and PowerPC architectures. All fetching is done in units of pages. When these pages reside in real memory, they are capable of being addressed by the CPU. These pages can also reside on disk but are obviously not addressable by the CPU when they are there. We said that a page fault causes data to be fetched into memory. More specifically, a page fault is a detection, accomplished by a hardware interrupt, that a virtual page cannot be addressed. This means that the page is not in real memory, and the virtual memory address cannot be translated into a real-memory address.

When this interrupt occurs, the VMM looks to see if there is any room in real memory in which to copy the data from the disk. In a moment, we will investigate how the VMM page stealer creates room in real memory, but for now, let's assume that there is enough room. The VMM examines the page to see if it has ever existed in the context of this process. If not, this is called an initial page fault for that page. An initial fault causes the VMM to allocate two different pages. One page is the real address in RAM where the page is to reside; the other page is the backing page on disk where the page will be saved if the page ever has to be removed temporarily from RAM. The concept of allocating only when the virtual page is first referenced is called *late page space allocation.* Later, we will contrast this with *early* page space allocation.

If the page has previously existed, that is, if it has an image of it somewhere on disk, the event is called a *repage* fault. The VMM looks up the disk address of this page in the external page table, finds its disk address, and schedules I/O for that page to be read from page space into RAM. The act of resolving a repage fault by copying the page from page space into RAM is called *page in.* The process that is blocked while waiting for a page in to complete is said to be in *page wait* state.

Now let's talk about that most larcenous of AIX system components—the page stealer. The job of the page stealer is to ensure that there is a small supply of free RAM pages available when an initial page fault occurs. When the number of free RAM pages dips below a certain value, the page stealer goes into action and attempts to steal pages to add to the free-page list. The page stealer stops when the number of free pages exceeds a different value. The certain value that spurs the page stealer into action is the minimum free-page number. The value that causes the page stealer to stop stealing is the maximum free-page number.

The page stealer really gets a bad rap. It should more properly be called a page borrower. The RAM pages are borrowed, or replaced, by selecting the stalest, or the *least recently used* (LRU), pages and somehow getting rid of them temporarily. One possibility is that a page is copied to page space if the page in RAM has been modified since it was last paged in. In that case, the page is called *dirty* and is *paged out* to page space. The other possibility is that the page is clean, which means that the copy in RAM matches exactly the copy in page space. A clean page does not need to be paged out and is simply purged.

I mentioned stealing pages to page space as the target *backing store* because page space is what is first associated with paging. Backing store is where dirty pages go when pages are stolen. There are two types of backing store: page space is backing store for working, or nonpersistent, pages, and a file system is backing store for persistent, or file, pages. To put it more simply, files page to and from filesystems, and everything else pages to page space.

Just to make sure you understand the performance burden of paging, let me summarize. Page faults do not necessarily cause disk I/O. Only repage faults cause page-in I/O. Page-out I/O occurs only when there is a shortage of free RAM pages and the page destined to be replaced is dirty. Thus we might consider that page-out I/O is the most direct measure of how constrained real memory is at a particular moment. Page-in I/O is a valid but less direct measure of constrained memory. Also note that a low amount of free memory is not an indication that memory is constrained. The system could quite happily run with free memory hovering just above the minimum free-page value. In fact, the system could be experiencing some initial page faults that could be resolved by stealing pages that are stale but clean, thus resulting in memory allocation activity with no accompanying I/O. In summary, nonzero paging rates are not good, but low (almost zero) free RAM is not necessarily bad. Later I will discuss the VMM role in filesystem I/O and will explain why free RAM pages tend to hover around a few hundred (this is what I mean by "almost" zero).

2.3.2 Late Page Space Allocation

Late allocation of the backing page is a unique feature of AIX. Let's contrast early and late page space allocation in order to understand the advantages of each. Let's say you wrote a program that dynamically allocated a very large array in storage and accessed elements of that array in a random fashion. Let's also assume that the data the program encountered today caused only 1% of the elements to be used. Early allocation would allocate 100% of the array on backing store, even though the program needed only 1% of that today. Late allocation would allocate only the 1% of the array on backing store as each page is needed. This might be compared to the old saying: Eat all you want, but take only what you can eat. That way there are no wasted "bytes"—either on your plate or on page space.

There is a trade-off to late allocation. It is possible that when it is time to allocate a backing page there is none to be had. AIX does a pretty ugly deed when there is a shortage of page space. It kills processes in an attempt to free up page space so that the system can continue to run. This is the "sacrifice of the few for the good of the many" approach, which is not always fair or wise. When page space first begins running low, a warning is

issued two ways. First, a message is posted to the system error log warning that a low condition has been reached. Then a message is posted to the system console warning that page space is low and processes should be terminated. If free-page space continues to decline, a kernel process is dispatched to conduct a "which" hunt. This kernel process scans the kernel process table looking for *which* process could be the best one to dispose of for the good of the system. The process that is chosen is not always the first process to encounter shortages, nor is it necessarily the process using the most storage. The victim is the youngest process. The chosen process is first sent a `SIGDANGER` signal. If this process did not take steps to catch `SIGDANGER`, then the process dies immediately. If the process does catch `SIGDANGER`, then the process can take remedial action. The best action is to save work and exit. If page space consumption continues beyond a level called the *page space kill* value, the hunter kernel process will issue a `SIGKILL` signal to the victim process. `SIGKILL` cannot be caught, and the victim process meets a grisly death.

Your first reaction to this severe action to page space shortage might be to gag in horror. How crude of AIX to kill processes indiscriminately. That isn't standard UNIX behavior. You would be partially right. The POSIX 1003.1 standard implies that `malloc` memory allocation system call should guarantee that storage be committed if the `malloc` call returns successfully. And, indeed, customers complained that this behavior did not comply with the standard. The POSIX standards body agreed with the customers and declared that AIX did not conform with the intent of the standard, even though AIX passed all the compliance tests.

In order to make AIX POSIX compliant to customers' satisfaction, the AIX developers fixed the behavior of the page space shortage algorithm. First, the default behavior is as previously described. If you want different behavior, you have to take explicit action. The action you take is to set the `PSALLOC` environment variable. Using Bourne shell syntax, this would look like

```
PSALLOC=early
```

If the variable is set to early, the process behavior is different. Most important, early page space allocation will occur for that process. The `malloc` call will return success and will have committed all the page space pages before returning to the caller. Furthermore, the process will be immune to the `SIGDANGER` and `SIGKILL` signals, so it cannot be killed for its page space usage sins.

You may be tempted to have the entire system use the POSIX behavior by setting `PSALLOC` in the `/etc/environment` file. Or you might set it in your shell *.profile* so that just your process hierarchy uses this behavior. I strongly recommend that you do not do this. There are many programs that greedily allocate much more storage than they actually use. The AIX X Window server is a good example of a program that allocates a 256-megabyte segment, but it typically uses 1% to 10% of that. Thus if you set `PSALLOC` in your .profile file, X Window would refuse to run; it could not satisfy its memory allocation because the free-page space was too small.

Even though paging space configuration is an important issue, tuning the amount of page space is never a performance issue. Stated another way, choosing too much or too little page space will not make your program run either faster or slower. However, too little page space may prevent your program from running at all.

2.3.3 Filesystem Caching

I mentioned earlier that the VMM plays a major part in the operation of the AIX filesystem. Since the VMM manages all virtual memory, the VMM also manages memory on behalf of the filesystem. All of RAM is shared among file pages and nonfile pages, and the VMM decides which pages occupy RAM at any given time.

Recall that the page stealer replaces pages in RAM based on an LRU algorithm. Until now we had been considering pages without regard to type of pages; however, it is useful to know that virtual pages are classified into three types, based on the location of backing store of the page. See Figure 2.2 for an illustration of this concept.

Pages are collected into a virtual segment that can be of one of three types: persistent, working, or client. Persistent segments represent a file in a filesystem and, therefore, can be backed by a filesystem. In other words, the pages are paged in and out of the filesystem. Working segments are backed by page space. Client segments represent a file in a mounted NFS filesystem and are, therefore, backed by a filesystem on an NFS server somewhere across the network. Since persistent segments and client segments are both a kind of filesystem-backed segment, it is also useful to think of segments as members of two broader categories: file segments and nonfile segments.

The VMM page stealer treats all three types of pages equally when things are quiet on the page-stealing front. I'll call this *quiet* condition the normal LRU page replacement situation, but increased paging activity causes the page stealer to treat file pages differently than nonfile pages. There are two situations involving nonquiet paging conditions. I'll call the *louder* case the "file pages only" LRU page replacement. In this case, file pages exceed a value called the maxperm threshold. (AIX uses this unfortunately inconsistent term to mean maximum file page limit. I suppose `maxperm` is short for maximum permanent.) When the number of occupied file pages exceeds this value, the page stealer chooses only file pages as replacement victims. This bias toward selecting file page victims is an attempt to limit the number of file pages. Note that this threshold is not a hard limit because it is possible for the number of file pages to drift beyond the threshold before the page stealer can be called into action.

The second *nonquiet* situation is when the number of file pages is below the maxperm value but above the `minperm` (minimum number of file pages) value. In this case the page stealer chooses which kind of page to steal by examining two other statistics. The VMM keeps count of repage rates for both file and nonfile pages. The page stealer will chose file pages if the file repage rate is greater than the nonfile repage rate. Otherwise the page stealer will treat file and nonfile pages equally.

The reason for all these decisions is that a system that experiences a lot of file I/O may tend to push out nonfile pages such as program text and data images. Program text images are persistent segments but are treated as nonfile segments for LRU purposes. It might sometimes improve overall system performance by limiting the amount of RAM that file pages occupy, thus keeping room for executable images. Otherwise the file page's greediness for memory could induce excessive page-in activity when the executable images are needed. In Chapter 4 I will discuss how you can change the `minperm` and `maxperm` values.

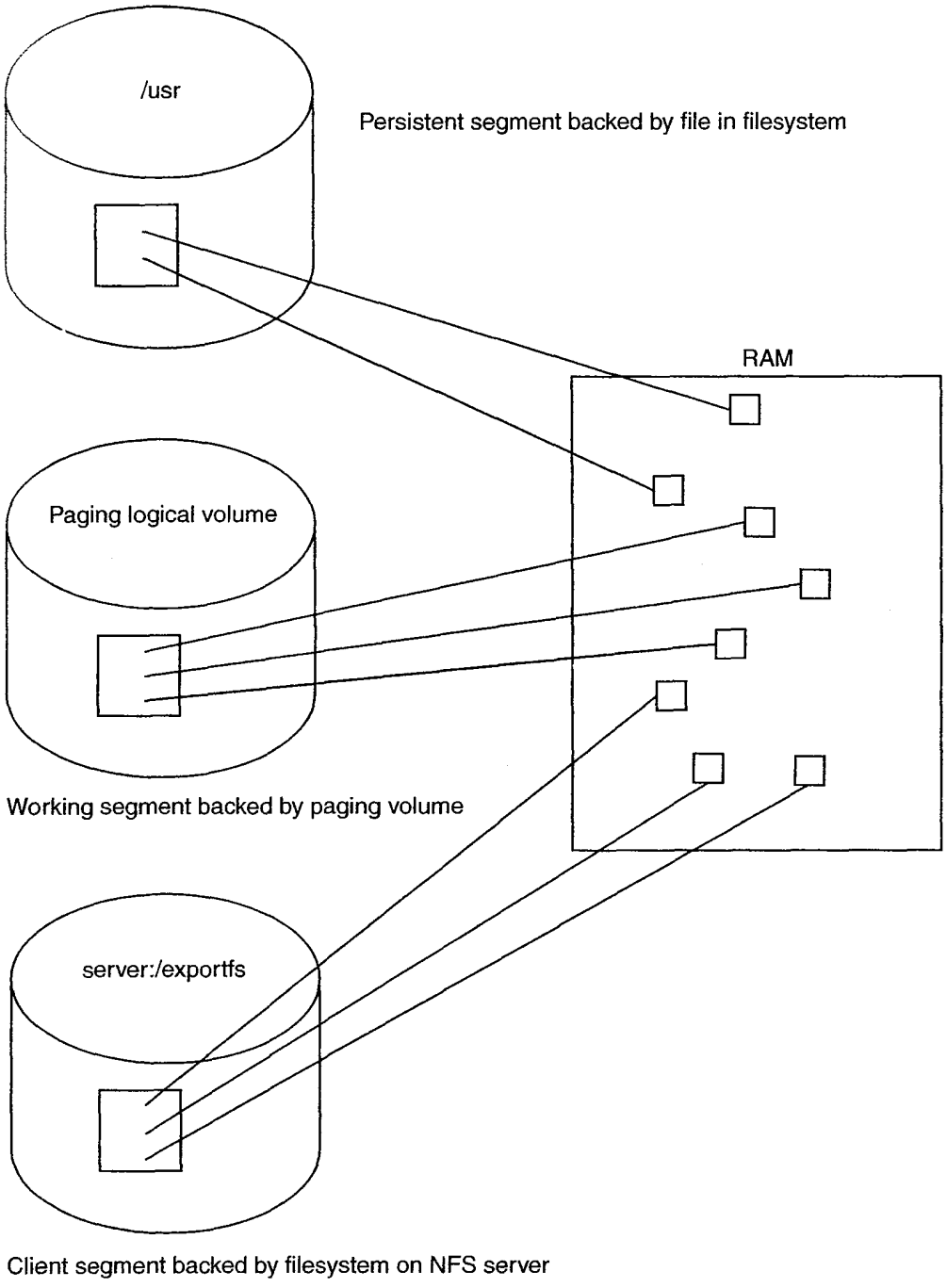


Figure 2.2 Virtual segment types

2.4 I/O Overview

A unique AIX feature contributing to AIX as an industrial-strength operating system is the Journal Filesystem (JFS). AIX was probably the first UNIX operating system to possess this feature, and others have recently followed. Let's discuss how the filesystem itself intertwines with the VMM and other system components.

2.4.1 Layers for File I/O

It is useful to view file I/O as four distinct layers. See Figure 2.3. It is useful to understand these layers because one of the performance statistics gathering tools that we will learn about in Chapter 3 tells how these layers are performing. The first layer is the *logical file.* Each read system call your program does is a logical read. However, each logical read does not necessarily result in a physical read. If the data desired is already contained in the VMM file cache, the VMM has no need to fetch the data from the disk, thus avoiding a physical read. The VMM manages the virtual segment layer. The next layer is managed by the logical volume manager (LVM), the genie of UNIX system administrators that responds to: I wish it was easy to partition disks for filesystems. LVM allows filesystems to span multiple disks and is responsible for translating file I/O requests into individual disk requests. The last layer is the physical volume layer, or device layer. This layer is managed by the disk device driver.

2.4.2 JFS Log

In a nutshell, JFS employs a journaling technique similar to a database in order to ensure consistency of the data making up the structure of the filesystem. This technique involves duplicating the meta-data transactions to the filesystem logical volume and to the journal. Filesystem meta-data is the data that makes up the structure of the filesystem itself, namely superblock, directories, inodes, and indirect data pointers. The journal, sometimes called the log, is another logical volume separate from the filesystem logical volume. It is used as a circular list in which to store the duplicate transactions. The journal is written to the disk before the filesystem meta-data is; thus the journal can be used to reconstruct the meta-data transactions if some of the transactions fail to complete. Frequent meta-data corruption would cause filesystems to become unusable in the days before journaled filesystems. This frequent corruption gave UNIX filesystems the reputation for lacking industrial strength.

Let's use an example to see in detail how the journal works on the JFS so you will appreciate what value it adds. Assume you wanted to create a new file.

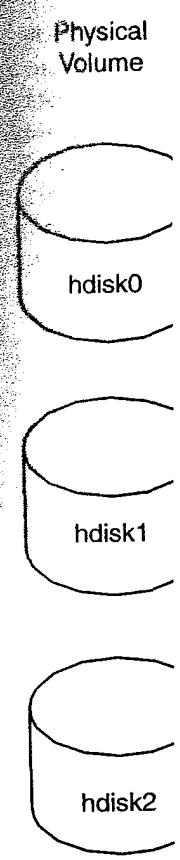
```
ps -ef >> ps.out
```

The system actions are

shell opens ps.out file

ps writes one new block at the end of the file

shell closes ps.out file



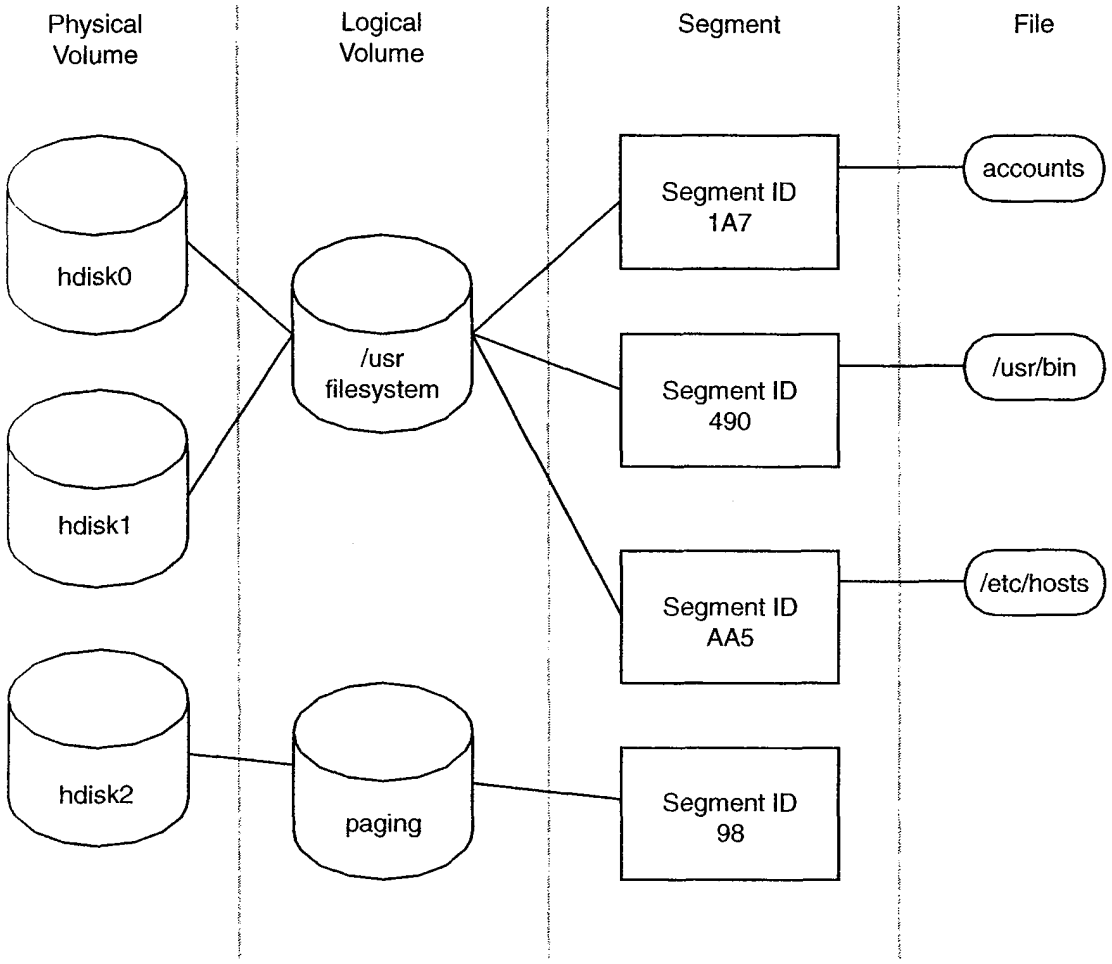


Figure 2.3 I/O system layers

After this command executes, the following meta-data is changed:

Current directory data is changed to add new entry for ps.out file.

Inode for current directory is changed to update modified time.

New inode created for file.

Free list changed to contain one less block.

These filesystem changes are performed twice. First the changes are recorded in the log by writing them in memory. The real data is recorded in the various memory structures that represent the disk meta-data. Figure 2.4 represents the state of the log after it has been written to disk. The state of the filesystem meta-data is shown after it was modified

in memory but before it was written to disk. By using AIX kernel lock synchronization, log data is guaranteed to be written to disk before the filesystem meta-data. This data is written in one atomic chunk in a compact contiguous spot on the disk; the filesystem meta-data is written in several chunks across the disk. When the meta-data is confirmed to be completely written, a pointer residing in the log header is updated to point after the end of the corresponding log data. Figure 2.5 represents the state of the journal and meta-data after the meta-data is written to disk. Note that the sync point has moved to signify that the meta-data has been committed to disk and that the corresponding portion of the log can be forgotten and is available to be written over.

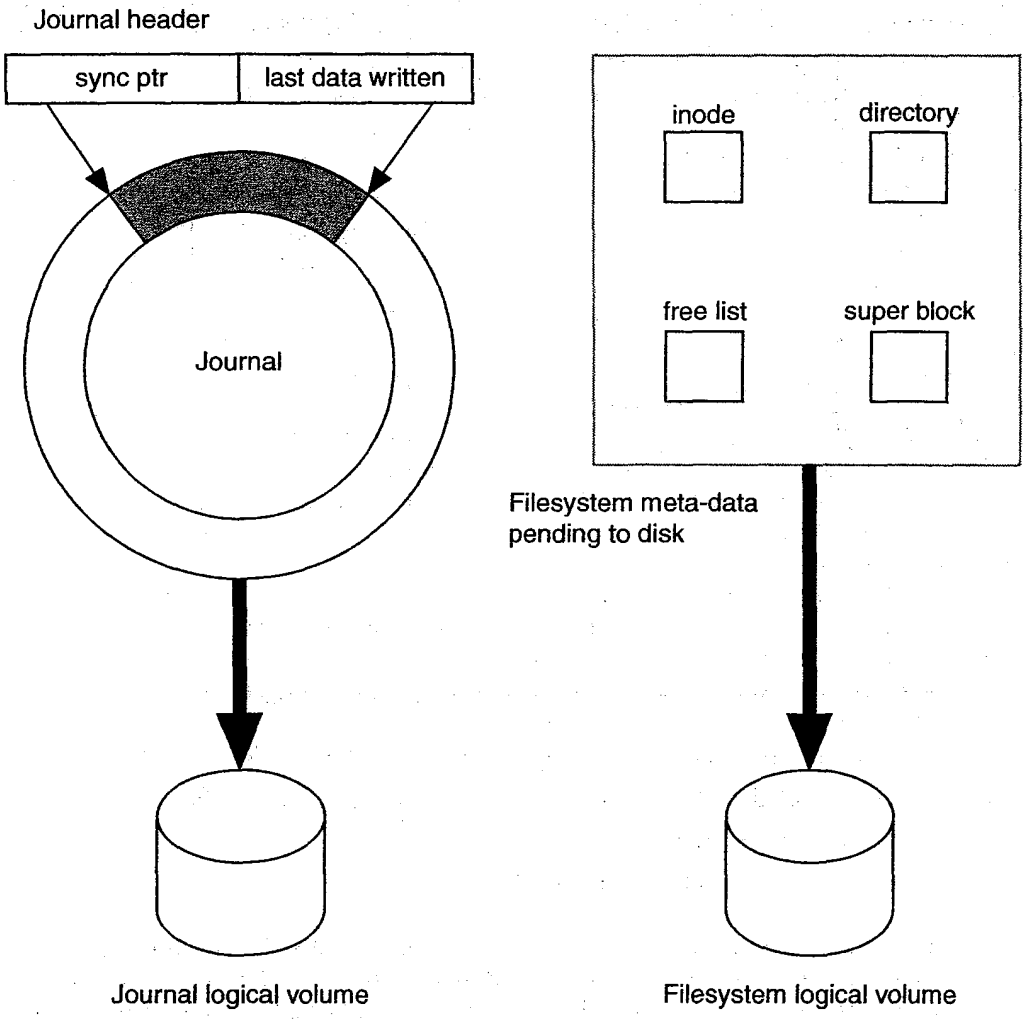


Figure 2.4 Journal before commit

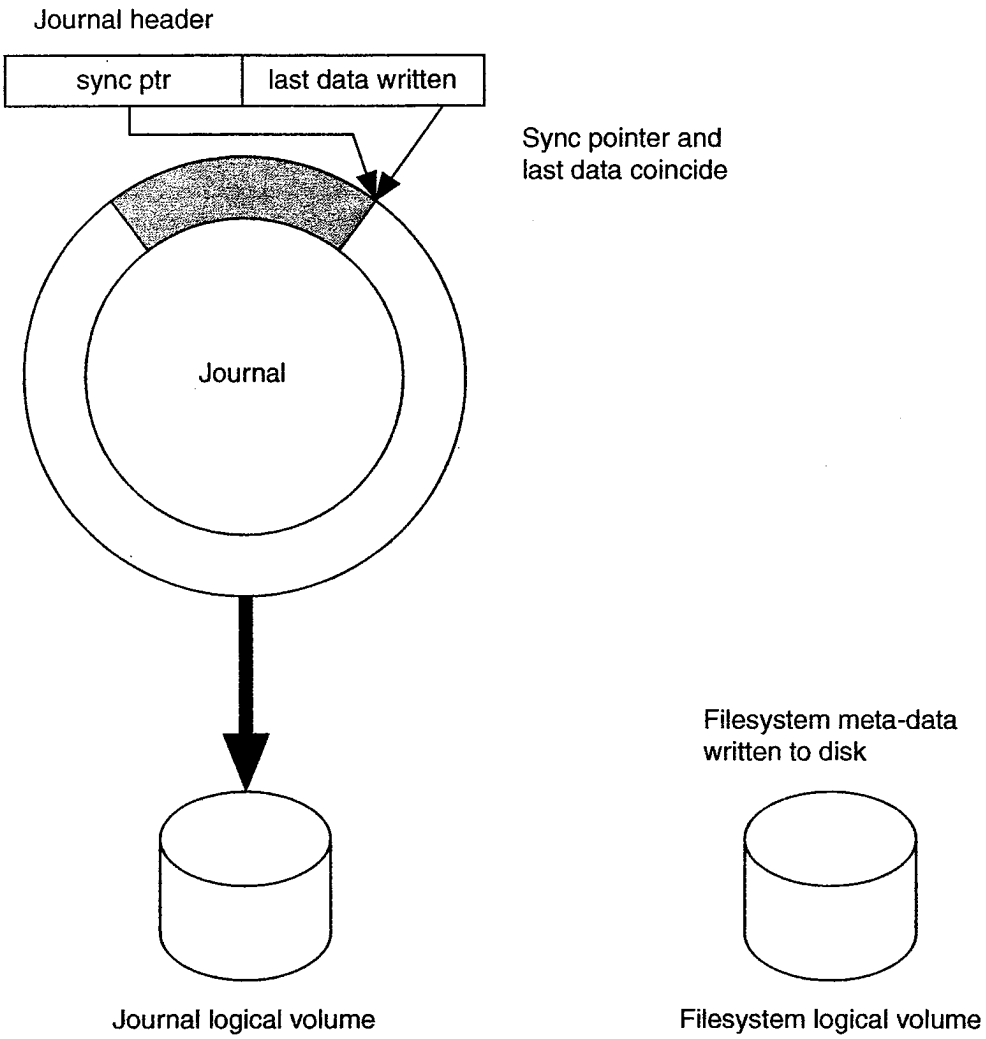


Figure 2.5 Journal after commit

If the system crashes between the times represented in Figures 2.4 and 2.5, the filesystem can be restored to a consistent state. When the system reboots, a program called *logredo* is run. It replays the journals to reconstruct the various corrupt filesystems. The logredo is run before the filesystems are mounted and takes the place of the archaic *fsck.* program.

The point of describing the journal mechanism is not to have you understand how the recovery works, but rather to show that extra disk I/O is being done to the journal. Usually this extra I/O is not noticed due to the low frequency. Note that the journal is not written if a file is read because the filesystem is not changing. Note, too, that the journal is not written if a file is being updated without appending new data.

2.4.3 Sequential Read-Ahead and Write-Behind

Programs that read files in a sequential manner can achieve greater throughput compared with performing the same amount of I/O in a random manner. This is due to a VMM feature called *read-ahead*. The VMM notices that a program has read two pages consecutively and anticipates that the program will continue to read consecutive pages. The anticipation results in VMM reading the pages before the program asks for them. When the program does read the pages, they will already be in RAM, and the program does not need to wait, thus reading more data more quickly.

The VMM performs this anticipation of sequential reading with the help of two thresholds—a starting read-ahead value (`minpgahead`) and an ending read-ahead value (`maxpgahead`). Upon detecting that two pages were read consecutively, the VMM will start by reading an additional number of pages specified by the `minpgahead` value. If the program continues to read sequentially, the VMM will keep doubling the read-ahead amount until the `maxpgahead` value limits the read-ahead size. If the program performs an *lseek* to the file to cause pages to be skipped, then read-ahead mode is canceled. Future detection of read-ahead mode on the same file would start from scratch. The defaults for `minpgahead` and `maxpgahead` are 2 and 8, respectively.

Where there is sequential reading going on, there is probably writing going on as well, and sequential writing throughput can be improved by some special processing. When a program writes data to a file, the modified pages will tend to sit in memory until a sync call by the sync daemon flushes the pages to the filesystem. However, if sequential writing is detected, flushing the dirty pages will happen early. This action is called *write-behind*. This detection is more primitive than that of read-ahead detection. When a program modifies all four pages of a 16K chunk and then proceeds to modify the next page, the previous four pages will be flushed to disk.

In Chapter 4 you will learn how to use vmtune to tune read-ahead and write-behind.

2.4.4 Logical Volume Manager

AIX has a unique feature of allowing filesystems to be managed dynamically without requiring the filesystem to be taken off-line. The AIX component that accomplishes this is called the Logical Volume Manager (LVM); it allows you to assign filesystems to chunks of disk called logical volumes. The LVM is implemented as a device layer between the VMM and the disk device driver. See Figure 2.6 for a conceptual picture of how the LVM fits into AIX.

The LVM itself does not affect disk I/O performance to any large extent. In fact, the extra layer probably adds a bit of overhead that is imperceptible in the overall scheme of things. However, to the system administrator seeking to optimize the system, the power of the LVM is in the flexibility and ease with which you can assign your data. On most UNIX systems, moving data from one disk to another or to another place on the same disk entails many hours of backing up, taking the system down to move disk partitions, rebooting, and copying backups to their new locations. On AIX, similar tasks involve changing some disk configuration parameters and telling the system to move the data for you.

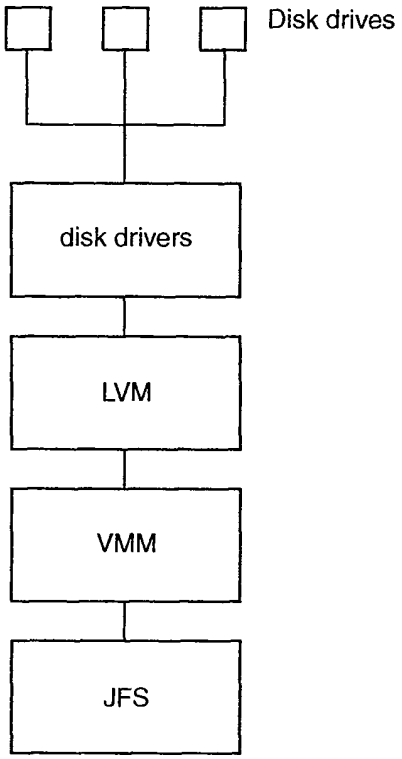


Figure 2.6 LVM hierarchy

I'll review some LVM terminology that is important to understand optimum data placement on disk. In Chapter 4, I'll discuss how to apply these concepts to learn how to control where your data goes.

See Figure 2.7 for a picture of the hierarchy of disk management objects. A disk (one that you could hold in your hand, if you knew how to extract it from your system) is called a physical volume (PV). A volume group (VG) is a collection of one or more PVs. A PV is divided into contiguous chunks called a physical partition (PP); each PP is nominally 4 megabytes but can vary from 1 to 256 megabytes. A VG can be grouped into several logical volumes (LV) that are divided into logical partitions (LP) that are the same size as the PPs. Logical volumes can span physical volumes, but they cannot span different volume groups.

When you create a logical volume (if you are creating a filesystem, a logical volume with sufficient size to contain the filesystem you specify is created for you), you must specify the volume group in which to place the logical volume. This specification limits which physical volumes the files can be on. Furthermore, you can specify that the logical volume be allocated in one of five bands on the disk. See Figure 2.8 for a picture of how these bands are laid out on a physical disk. The total number of cylinders is divided into five equal-sized bands of 20% each. From the center axle, the bands are called *inner edge*,

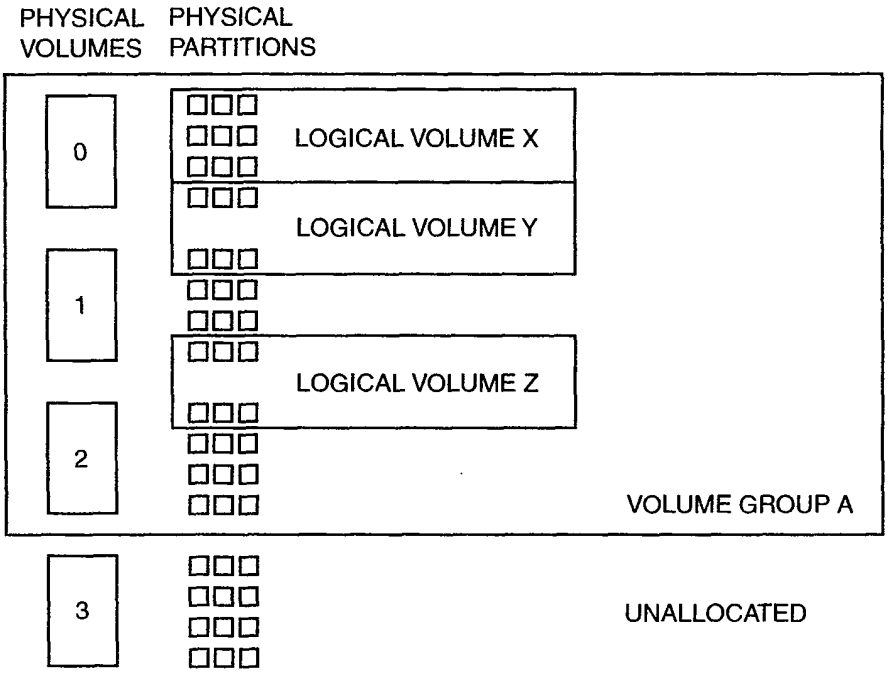


Figure 2.7 LVM object terminology

inner middle, center, outer middle, and outer edge. The band of optimum seek time is the center. A file residing on the center cylinder will have an average seek distance of one-half the total number of cylinders. All other file positions will have average seek distances greater than one-half.

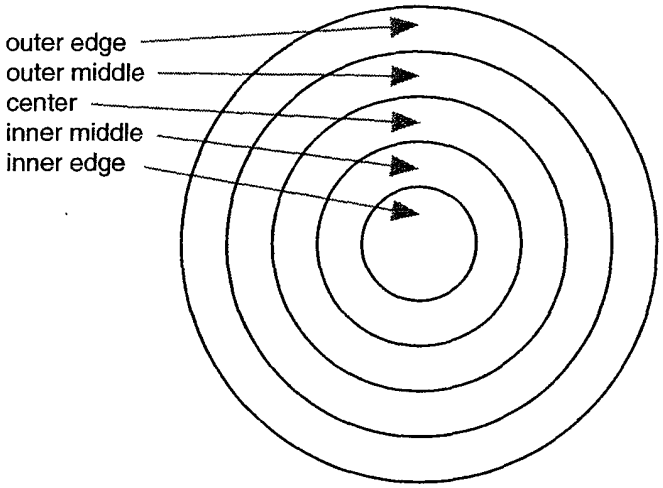


Figure 2.8 Disk allocation bands

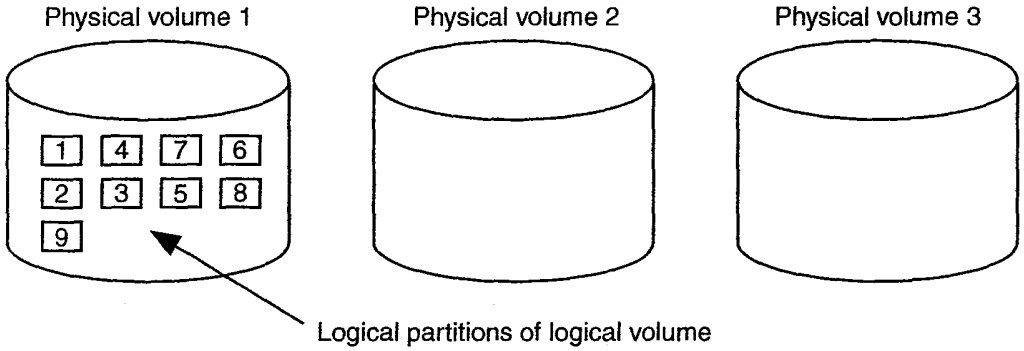


Figure 2.9 Minimum volume allocation for multivolume group

Since a logical volume can exist in a volume group that contains more than one physical volume, there is an LVM rule that decides how many physical volumes to use when allocating the partitions for the logical volume. The default is *minimum*, which tries to use the least number of volumes. Figure 2.9 shows how this would look for an LV with nine partitions. The alternative allocation, *maximum*, is to spread the logical partitions across as many physical volumes as possible. This arrangement, shown in Figure 2.10, might be advantageous on a system where there are many independent processes accessing the same filesystem. Since different physical disks can be executing their seek operations in parallel, there is more opportunity for parallel I/O operations with maximum. However, for a single process accessing a filesystem, maximum and minimum are about the same. The single process would block if one of the disks needed to be read with no opportunity to put the other physical volumes to work. With an ordinary JFS filesystem, it takes multiple processes to put multiple disks to work. However, a single process can put multiple disks to work concurrently if it is performing sequential I/O on a striped filesystem. We will investigate striped filesystems in Chapter 4.

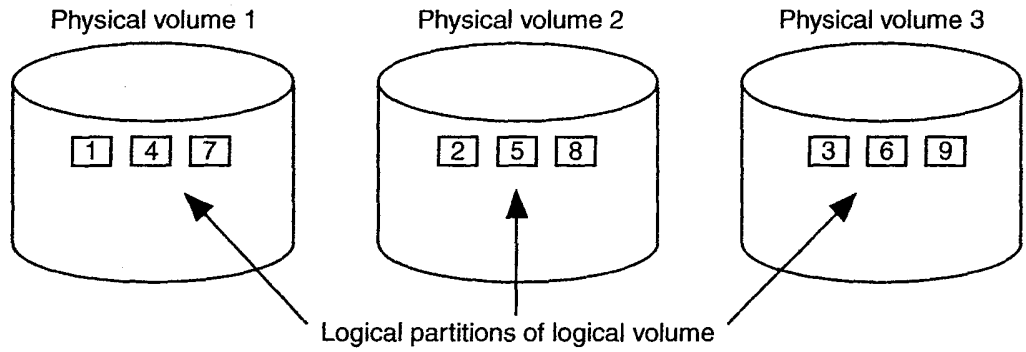


Figure 2.10 Maximum volume allocation

2.5 CPU Scheduling Overview

Effective use of the hardware CPUs (remember that a system may have one or more CPUs) is a job for the AIX scheduler. There are two situations to understand: The first is normal thread scheduling which governs how work is handled in most situations; the second is when memory usage is abnormally high and the system is virtual memory thrashing.

2.5.1 Scheduling Threads

Before I discuss how the CPU (or CPUs in the case of multiprocessor hardware) is managed by the AIX kernel, I had better define a *thread*, which is the scheduling entity for AIX Version 4. For AIX Version 3, the scheduling entity is the process which the `fork` system call creates. After a fork, parent and child processes share a copy of the program-executable text, but they receive their own copy of the program data. Therefore parent and child cannot communicate by modifying variables in the data segment but must resort to some interprocess communication means. New to Version 4 is the concept of threads, which is different from a process in that a child thread does not get a new copy of the data and therefore can communicate with the parent thread by modifying program variables. There are other subtle differences between threads and processes, but I won't discuss them here. I suggest you consult *Pthreads Programming* by Bradford Nichols and others for more information about threads.

If you are using an AIX Version 3 system, you can substitute *process* for *thread*, and the meaning will be the same.

Since there are many more threads than there are CPUs to run them on, the operating system decides which thread at any moment can use the CPU. The AIX scheduler is the component burdened with this task—so many threads and so little CPU time. The scheduler chooses the thread from a list of eligible threads that wait in the run queue. The run queue is sorted in priority order, and the top priority runnable threads get to use the CPUs. You may not be accustomed to thinking of more than one CPU, but a multiple CPU configuration is the most general case.

Figure 2.11 illustrates one run queue serving multiple CPUs. A process is placed on the run queue at the level that corresponds to the thread priority, once the thread is awakened from sleeping on an event. Unlike the grocery checkout queue, where breaking into the middle of the queue is met with looks that would melt the offender's ice cream carton, breaking into the middle of the queue on AIX is normal and quite civilized. The new run queue entrant thread jumps to its rightful place in line, using its priority number like a special pass. Every thread has a priority number, ranging from 0 to 127. The best priority is 0, while the worst is 127. Because the adjectives "high" and "low" get confusing, I will consistently use "better" and "worse" to describe priority goodness.

Thread priorities can be either fixed forever or changed as a function of time spent using the CPU. Most user processes have changing priorities, while most system kernel processes have fixed priorities. Threads that use the CPU are penalized by changing their priority to be worse; threads that are waiting to use the CPU are rewarded by changing their priority to be better. Let's examine exactly how this happens in detail, because in Chapter 4 you will learn how to affect the priority calculation.

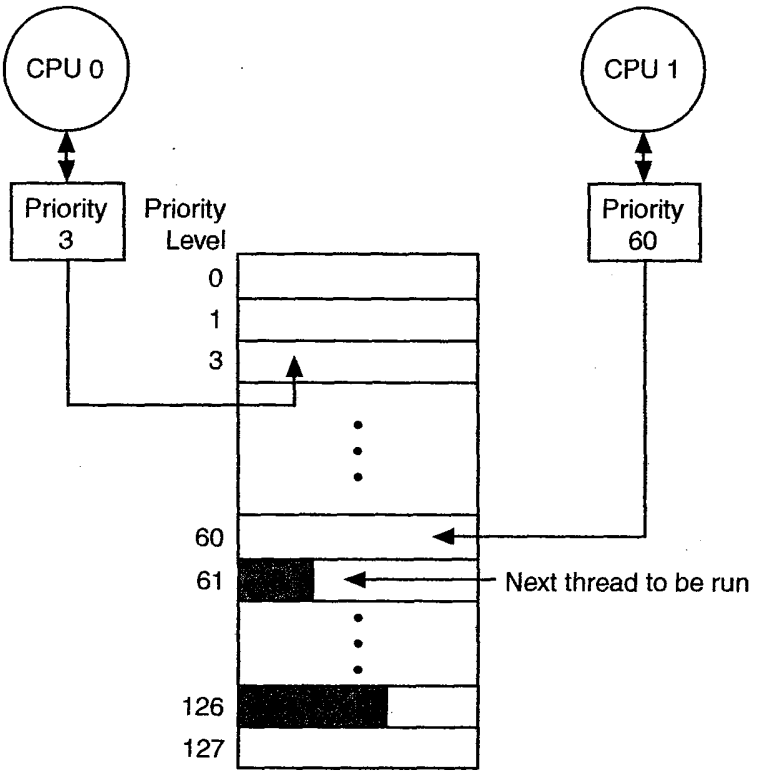


Figure 2.11 Single run queue serving multiple CPUs

The threads that are running on the CPU get a new priority calculation 100 times per second, based on the system 100 Hz clock interrupt. This can be expressed in the following formula:

```
CPUnew = min (CPU old +1, 120)
```

CPUnew is the new value of the CPU penalty value that is calculated by adding 1 to the old CPU penalty. The CPU penalty cannot exceed 120. Loosely interpreted, this formula says that the running thread has its CPU penalty increase until the maximum of 120 is reached. If the penalty is zero, it would take 1.2 seconds until the CPU penalty reached the maximum.

Once every second, all threads, including those that are asleep, have their CPU penalty calculated with the following formula:

```
CPUnew = CPUold*D/32  where D=16 by default
```

D is the CPU penalty decay factor. If we use the default value of 16 for D, then the CPU penalty value will be halved every second. If the penalty value were 120, it would take seven seconds for the value to be degraded to zero.

The priority that the scheduler uses is calculated with the following formula:

```
P = 40 + N + (CPUnew * R)/32  where R=16 by default
```

P is the priority value; N is the nice value. It is set typically to 20 and can be changed with the nice command. C will eventually grow to the maximum value of 120; R is a weighting factor and is defaulted to 16.

Let me summarize these formulas. While a thread executes, its CPU penalty increases; therefore its priority increases or becomes worse. The thread is penalized for running. While a thread sleeps, its CPU penalty decreases; therefore its priority decreases or becomes better. A thread is rewarded for sleeping. When the sleeping thread wakes up, it is put on the run queue with the same priority that was computed while it was sleeping.

Competing threads on the run queue get a fair opportunity to run because the priority of a running thread increases while the priority of a runnable, but not running, thread decreases. Every T ticks. Where T is typically one, the scheduler compares priorities of the running threads against those in the run queue and preempts them if a waiting runnable thread has a priority greater than that of the running thread.

Preemptive dispatching could be compared to a car repair shop with one mechanic, the owner. You bring your car (process) into the garage for a valve job. This takes a very long time to complete. The mechanic works on the car for an hour or so; another customer comes in for an oil change and wants to wait. The mechanic considers this job a high priority task (interactive job). He suspends the valve job and begins work on the oil change, completing it in 30 minutes. After the customer with the oil change leaves, the mechanic returns to the valve job.

2.5.2 Memory Overcommitment

The AIX scheduler gets called into action once again when processes are using too much virtual memory. This overcommitment of virtual memory is measured by the amount of paging that occurs. When paging exceeds a certain threshold, processes are suspended in an attempt to throttle back the memory load. (I know I said earlier that threads are equivalent to processes, but in this case I mean processes and all the threads contained therein.) While processes are suspended, no new forks can occur. When the paging situation returns to normal, suspended processes are reactivated. Once reactivated, a process is exempt from being suspended for a short time. Certain other processes such as kernel processes and fixed priority processes are always exempt from suspension.

Please understand that if there is no paging activity there is no problem. If there is a lot of paging, programs near and dear to you may stop executing.

Let's examine just what I mean by "a lot of paging." Imagine a page getting paged out by the page stealer and immediately being needed by some process. Imagine this happening with every page that gets paged out. If paging was ever this bad, the VMM would spend all the CPU time trying to resolve page faults of pages just paged out. The system going over the edge of sanity and spending more time paging than doing anything else is called *thrashing*. Let's come up with a formula that quantifies how much thrashing is tolerable.

I said earlier that page stealing does not always result in a page out. With luck, the page is not dirty, and stealing it means simply reusing the page. With no luck at all, every

page would be dirty, and every page steal would result in a page out. Let's now define a thrashing severity ratio

```
T = O/S
```

where S is the number of page steals in the last second and O is the number of page outs to page space in the last second. T cannot be larger than 1, and 0 is ideal. However, a value of 1 would mean a useless system. There is a value of T that when T > H, the AIX scheduler blows the whistle and considers putting processes into the penalty box. In this case the penalty box is a thrashing suspension queue. You can tune the value of H, which has been empirically set to 1/6 by default. You can adjust H to suit your opinion of when thrashing is bad enough to suspend processes.

Now that we know when thrashing occurs, we need to know which processes are chosen as suspension victims. If a process appears to be the main cause of all this paging, it should be suspended. Every process has statistics recording how many page faults it incurred in the last second, and of those faults, how many were repage faults. In the AIX justice system, suspicion of wrongdoing is grounds for conviction. Consider the ratio

```
X = R/F
```

where R is repage faults by a certain process in the past second, and F is the total faults in the last second. When X > P, that process is sentenced for some time in the penalty queue. P has a default value of 1/4.

Once processes are suspended, the time during which the thrashing severity must return to normal is the value w, which has the default value of 1. The default value of m is 2. If there are less than m processes, suspension will not occur. Once a process is reactivated, it will be given a grace period in which it is exempt from suspension. This grace period, e, has a default value of two seconds.

In Chapter 4 you will learn to tune values of p and h, reciprocals of P and H, respectively, and e, m, and w.

2.6 Facts of Life about Disk Hardware

In order to understand how to maximize general system I/O performance, it is necessary to understand the limitations of random access SCSI hard disk technology. The disks of today are made of pizza-like platters spinning at 3000 to 8000 rpm. The data is read from a magnetic device on the end of a set of mechanical arms that move from outside edge to inside edge of the platter. These disks can deliver data to RAM in a typical time of 10 to 50 milliseconds. On the other hand, the CPU can access data in RAM in an average time of 20 to 70 nanoseconds. To put it another way, disks are 1000 times slower than RAM. There are things that can be done to reduce the access time of the disk to 10 milliseconds.

There are four components of this access time. The largest component is the time required to move the mechanical arm from where it is to where the desired data is. This component is called *seek latency.* Seek latency for modern disk technology is somewhere between 10 ms and 100 ms. Once the arm has moved to the desired spot, the next largest time component comes into play. *Rotational latency* is the time required for the desired

data to pass under the mechanical arm. A typical disk spins at the rate of 7600 rpm. Thus a worst-case rotational latency would be 60/7600, or about 8 ms. Once the data passes under the magnetic head, it must be copied from the bits on the magnetic platter onto the wires attaching the disk to the SCSI controller card, which in turn attaches to the system I/O bus. This is expressed as the disk transfer rate. Disk transfer rates are between 1 and 10 MB per second. For a typical disk transfer of 4K on a disk with a transfer rate of 4 MB per second, the transfer latency would be 1 ms. After the data gets on the disk cable, it is transferred through the SCSI adapter through the system bus into RAM. This component is called the SCSI controller transfer rate. The rate for SCSI-2 is on the order of 10 MB per second; for SCSI Fast-and-Wide adapters, this rate is about 20 MB per second.

It is important to note that mechanical latency components are serial in nature. This means that seek latency cannot be overlapped with rotational latency. Nor can either of these be overlapped with disk transfer latency. While the disk is seeking to the next spot, that same disk obviously cannot be transferring any data. Since the disk platter spins at constant speed, the disk mechanism cannot even begin to wait for the platter to be positioned at the correct spot until the arm is done moving. Thus all the mechanical latencies are additive on the same disk. The SCSI controller transfer and the disk transfer can be overlapped up to the point where the SCSI transfer rate (more properly called bandwidth, but for our purposes I will declare them synonyms) is being exceeded. For example, it works fine to hook up two disks, each having a transfer rate of 3 MB per second, to an SCSI controller that has a transfer rate of 10 MB per second. However, connecting four such disks to the same controller is likely to induce a bottleneck, especially if the disks are actually approaching their maximum transfer rate for extended periods.

For some modern disks, the outer tracks have more sectors (a sector is the smallest unit of data that can be read from the disk) per track than the inner sectors. This is just a geometric fact that is exploited to increase the capacity of the disk. To us performance analysts, this is interesting because the more sectors per unit area, potentially the more bytes per second flying past the read head. And, empirical measurements show that the outer tracks on a variable density disk have a 15% to 30% improved transfer rate under optimum conditions.

The way to minimize the latency for one disk is to try to minimize each component. The only components that can be minimized are seek latency, rotational latency, and disk transfer latency, which is accomplished by judicious or lucky placement of your data on the disk. The SCSI controller transfer rates are fixed and cannot be changed. However, you can add additional SCSI controllers to boost total SCSI bandwidth to match the total bandwidth of the disks you have. We will revisit latency and transfer rates in Chapter 4 when we discuss how to optimize data placement.

2.7 Summary

The following are the most important points covered in this chapter:

1. The AIX kernel tables do not need to be tuned by recompiling the kernel. The kernel dynamically allocates internal kernel data structures that require no tuning.

2. The AIX program loader gets a program started by resolving symbolic addresses, mapping the portions of the program into its virtual memory space, and transferring control to the program. A knowledge of the program loader will enable you to understand how to optimize virtual memory usage.
3. The AIX VMM maps a program's virtual memory in the system's real memory. It handles page in and page out of files, program data, and program code. A working knowledge of the VMM will enable you to optimize disk I/O performance and virtual memory usage.
4. The AIX Journal Filesystem (JFS) is the subsystem that interacts closely with the VMM in performing file I/O. Knowledge of the inner secrets of the filesystem operation will enable you to optimize disk I/O performance.
5. The AIX CPU scheduler controls which threads run on the various CPUs of the system. Understanding of the scheduling rules will enable you to optimize thread contention for the CPU in CPU-bound situations.
6. Understanding the realities of current magnetic disk technology will enable you to place data on disks to optimize performance and to make configuration choices that perform efficiently.