This chapter covers the following key topics:

- Operating Systems Basics
- IOS Architecture Overview
- Memory Organization
- IOS Processes
- IOS Kernel
- Packet Buffer Management
- Device Drivers

CHAPTER 1

Fundamental IOS Software Architecture

If you were naming the most popular and widely used computer operating systems, which ones would you choose? Most likely, your list would contain names like UNIX, MS-DOS, Microsoft Windows, or even IBM's MVS for mainframes. These are all well-known operating systems—you might even be using one on a computer at home. Now, think for a minute; are there any others? Would your list contain Cisco IOS? No, it probably wouldn't, even though IOS is one of the most widely deployed operating systems in use today.

Unlike the general-purpose operating systems just mentioned, many people never encounter IOS directly. Most who use a computer to access the Internet aren't even aware IOS is behind the scenes. Even those who are aware of IOS, people who use it directly, often don't consider it to be an operating system but instead just the software that runs Cisco routers.

IOS might not run word processors or accounting applications like others on the list but it is still, in fact, an operating system—albeit, one specialized for switching data packets. As you will see, much of the IOS architecture is focused on switching packets as quickly and efficiently as possible.

Although IOS is constructed of many of the same fundamental components found in general-purpose operating systems, the components often contain key differences due to the design goals for IOS. This chapter covers these fundamental operating system components—the *software infrastructure* of IOS—and explores the rationale behind their design.

This chapter begins by introducing a few basic operating system concepts and terms that are useful in understanding the IOS architecture. If you already have a thorough understanding of operating systems, you might want to skip this first section and continue with the section, "IOS Architecture Overview." The remainder of this chapter deals with the major elements of the IOS architecture.

Operating Systems Basics

Modern operating systems provide two primary functions: hardware abstraction and resource management. Hardware abstraction gives software developers a common interface between their application programs and the computer's hardware so each individual programmer doesn't need to deal with the hardware's intricacies. Instead, the hardware-specific programming is developed once, in the operating system, and everyone shares.

The second function that operating systems provide is managing the computer's resources (CPU cycles, memory, and disk drive space, for example) so they can be shared efficiently among multiple applications. Like hardware abstraction, building resource management into the operating system keeps each application programmer from writing resource management code for every program.

CPU Resource Management and Multitasking

Although some operating systems only allow one program to run at a time (most versions of MS-DOS operate this way, for example), it's more common to find operating systems managing multiple programs concurrently. Running multiple programs at once is called *multitasking* and operating systems that support it are typically called *multitasking operating systems*.

Computer programs written for multitasking operating systems themselves often contain multiple independent tasks that run concurrently. These little subprograms are called *threads* because they form a single thread of instruction execution within the program. Threads each have their own set of CPU register values, called a *context*, but can share the same memory address space with other threads in the same program. A group of threads that share a common memory space, share a common purpose, and collectively control a set of operating system resources is called a *process*. On operating systems and CPUs that support virtual memory, each process might run in a separate address space that is protected from other processes.

Because a processor can execute instructions for only one program at a time, the operating system must manage which set of program instructions (which thread) is allowed to run. Deciding which process should run is called *scheduling* and is usually performed by a core piece of the operating system called the *kernel*. An operating system can use one of several methods to schedule threads, depending on the type of applications the operating system has been optimized to support. Different types of applications (batch, interactive, transactional, real-time, and others) have different CPU utilization characteristics, and their overall performance is affected by the scheduling method used.

The simplest scheduling method is to assign each thread to the processor in the order its run request is received and let each thread run to completion. This method is called *FIFO* (first-in, first-out) *run-to-completion* scheduling. FIFO's advantages are: It is easy to implement,

it has very low overhead, and it's "fair"—all threads are treated equally, first come, first served.

FIFO with run-to-completion scheduling is good for batch applications and some transactional applications that perform serial processing and then exit, but it doesn't work well for interactive or real-time applications. Interactive applications need relatively fast, short duration access to the CPU so they can return a quick response to a user or service some external device.

One possible solution for these applications is to assign priorities to each thread. Threads with a critical need for fast access to the CPU, such as real-time threads, can be assigned a higher priority than other less critical threads, such as batch. The high priority threads can jump to the head of the line and quickly run on the CPU. If multiple threads are waiting with the same priority, they are processed in the order in which they're received (just like basic FIFO). This method is called *run-to-completion priority scheduling*.

Run-to-completion priority scheduling, though an improvement over FIFO, still has one drawback that makes it unsuitable for interactive and real-time applications—it's easy for one thread to monopolize the processor. High priority threads can get stuck behind a long-running, low-priority thread already running on the processor. To solve this problem, a method is needed to temporarily suspend or *preempt* a running thread so other threads can access the CPU.

Thread Preemption

Involuntarily suspending one thread to schedule another is called *preemption*. Scheduling methods that utilize preemption instead of run to completion are said to be *preemptive*, and operating systems that employ these methods are called *preemptive multitasking* operating systems. Preemption relies on the kernel to periodically change the currently running thread via a *context switch*. The trigger for a context switch can be either a system timer (each thread is assigned a time slice) or a function call to the kernel itself. When a context switch is triggered, the kernel selects the next thread to run and the preempted thread is put back in line to run again at its next opportunity based on the scheduling method being used.

NOTE A *context switch* occurs when an operating system's kernel removes one thread from the CPU and places another thread on the CPU. In other words, context switches occur when the computer changes the task on which it is currently working. Context switches can be quite expensive in terms of CPU time because all of the processor's registers must be saved for the thread being taken off the CPU and restored for the thread being put on the CPU. The context is essential for the preempted thread to know where it left off, and for the thread being run to know where it was the last time it ran.

There are several advantages to preemptive multitasking, including the following:

- **It's predictable**—A thread can, within limits, know when it will likely run again. For instance, given the limits of kernel implementations, a thread can be set up to run once a second and the programmer can be reasonably certain that the thread will be scheduled to run at that interval.
- **It's difficult to break**—No single thread can monopolize the CPU for long periods of time. A single thread falling into an endless loop cannot stop other threads from running.

Of course, there are also disadvantages to preemptive multitasking, such as:

- **It's less efficient than run-to-completion methods**—In general, preemptive multitasking systems tend to switch contexts more often, which means the CPU spends more time scheduling threads and switching between them than it does with run to completion.
- **It adds complexity to application software**—A thread running on a preemptive system can be interrupted anywhere. Programmers must design and write their applications to protect critical data structures from being changed by other threads when preempted.

Memory Resource Management

Operating systems also manage the computer's memory, typically dividing it into various parts for storing actual computer instructions (code), data variables, and the *heap*. The heap is a section of memory from which processes can allocate and free memory dynamically.

Some operating systems provide a means for processes to address more memory than is physically present as RAM, a concept called *virtual memory*. With virtual memory, the computer's memory can be expanded to include secondary storage, such as a disk drive, in a way that's transparent to the processes. Operating systems create virtual memory using a hardware feature, available on some processors, called a *memory map unit* (MMU). MMU automatically remaps memory address requests to either physical memory (RAM) or secondary storage (disk) depending on where the contents actually reside. The MMU also allows some address ranges to be protected (marked read-only) or to be left totally unmapped.

Virtual memory also has another benefit: In operating systems that support it, an MMU can be programmed to create a separate address space for each process. Each process can have a memory space all to itself and can be prevented from accessing memory in the address space of other processes.

Although it has many benefits, virtual memory does not come for free. There are resource requirements and performance penalties—some of them significant—associated with its use. For this reason, as you will see, IOS does not employ a full virtual memory scheme.

Interrupts

Operating systems usually provide support for CPU interrupts. Interrupts are a hardware feature that cause the CPU to temporarily suspend its current instruction sequence and to transfer control to a special program. The special program, called an *interrupt handler*, performs operations to respond to the event that caused the interrupt, and then returns the CPU to the original instruction sequence. Interrupts often are generated by external hardware, such as a media controller requesting attention, but they also can be generated by the CPU itself. Operating systems support the interrupts by providing a set of interrupt handlers for all possible interrupt types.

IOS Architecture Overview

IOS was originally designed to be a small, embedded system for early Cisco routers. At that time, routers themselves were viewed mostly as hardware appliances—little distinction was made between the hardware and software. In fact, initially IOS wasn't even called "IOS"; it was just referred to as "the OS" that ran a Cisco router.

As routed networks gained popularity, demand rose for routers to support an increasing number of protocols and to provide other functionality, such as bridging. Cisco responded to this enormous demand by adding new features into the router software, resulting in the multi-functional routing and bridging software IOS is today. Interestingly, although the functionality of IOS has grown considerably, the basic operating system architecture has remained mostly the same.

Compared to other operating systems, IOS has a fairly simple architecture. Like most small, embedded systems, IOS was designed to be lean and mean to stay within the memory and speed constraints of the original platforms.

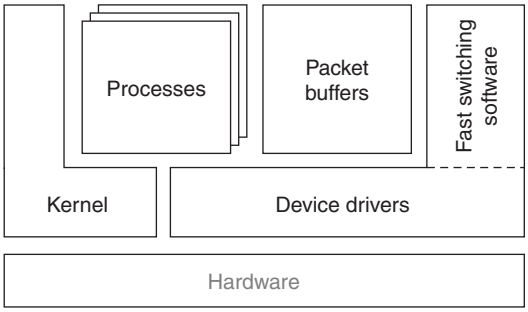
Early routers had limited amounts of memory to share between the software and data (such as routing tables). To limit the size of the executable image, IOS was designed to provide only essential services.

Speed was also a major consideration in the design. To maximize the router's capability to quickly switch packets, a conscious effort was made to design the operating system with a minimum of operational overhead and to allow maximum CPU bandwidth for packet

switching. Many safeguards, such as inter-thread memory protection mechanisms, found in other operating systems are missing from IOS because of the CPU and the memory overhead they introduce. In general, the IOS design emphasizes speed at the expense of extra fault protection.

Figure 1-1 shows a conceptual diagram of the IOS architecture.

Figure 1-1 *IOS Architecture*



As Figure 1-1 illustrates, IOS has five major elements:

- **Processes**—Individual threads and associated data that perform tasks, such as system maintenance, switching packets, and implementing routing protocols.
- **Kernel**—Provides basic system services to the rest of IOS, such as memory management and process scheduling. It provides hardware (CPU and memory) resource management to processes.
- **Packet Buffers**—Global memory buffers and their associated management functions used to hold packets being switched.
- **Device Drivers**—Functions that control network interface hardware and peripherals (such as a flash card). Device drivers interface between the IOS processes, the IOS kernel, and the hardware. They also interface to the fast switching software.
- **Fast Switching Software**—Highly optimized packet switching functions.

Each of these elements, except fast switching software, is discussed in more detail in the following sections. The fast switching software is discussed later in Chapter 2, “Packet Switching Architecture.” Before we investigate these architectural elements, let’s first look at how IOS organizes memory.

Memory Organization

IOS maps the entire physical memory into one large flat virtual address space. The CPU's MMU is used when available to create the virtual address space even though IOS doesn't employ a full virtual memory scheme. To reduce overhead, the kernel does not perform any memory paging or swapping, so virtual address space is limited to the bounds of the physical memory available.

IOS divides this address space into areas of memory called *regions,* which mostly correspond to the various types of physical memory. For example, SRAM might be present for storing packets and DRAM might be present for storing software and data on a given type of router. Classifying memory into regions allows IOS to group various types of memory so software needn't know about the specifics of memory on every platform.

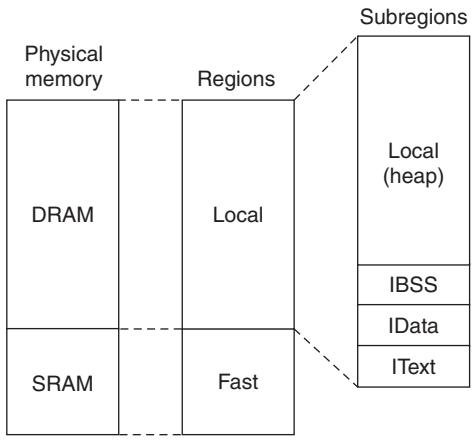
Memory regions are classified into one of eight categories, which are listed in Table 1-1.

Table 1-1 *Memory Region Classes*

Memory Region Class	Characteristics
Local	Normal run-time data structures and local heaps; often DRAM.
Iomem	Shared memory that is visible to both the CPU and the network media controllers over a data bus. Often is SRAM.
Fast	Fast memory, such as SRAM, used for special-purpose and speed-critical tasks.
IText	Executable IOS code.
IData	Initialized variables.
IBss	Uninitialized variables.
PCI	PCI bus memory; visible to all devices on the PCI buses.
Flash	Flash memory. This region class can be used to store run-from-Flash or run-from-RAM IOS images. It often also can be used to store backups of the router configuration and other data, such as crash data. Typically, a file system is built in the Flash memory region.

Memory regions also can be nested in a parent-child relationship. Although there is no imposed limit on the depth of nesting, only one level is really used. Regions nested in this manner form *subregions* of the parent region. Figure 1-2 shows a typical platform virtual memory layout and the regions and subregions IOS might create.

Figure 1-2 *Memory Regions*



The IOS EXEC command **show region** can be used to display the regions defined on a particular system as demonstrated in Example 1-1 (taken from a Cisco 7206 router).

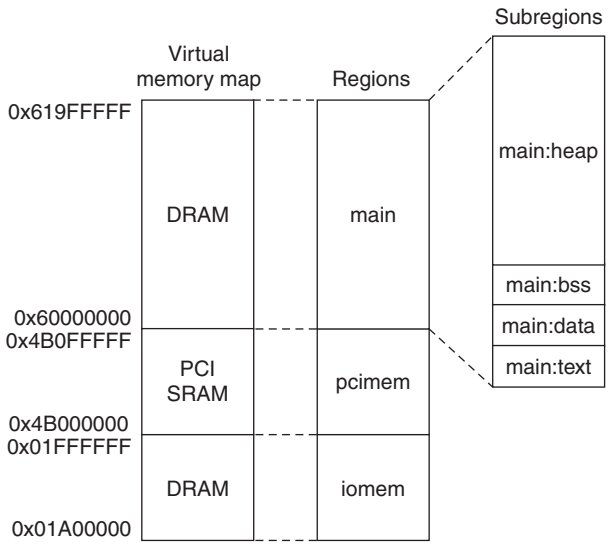
Example 1-1 **show region** *Command Output*

```
router#show region
Region Manager:
      Start         End     Size(b)  Class  Media  Name
 0x01A00000  0x01FFFFFF     6291456  Iomem  R/W    iomem
 0x31A00000  0x31FFFFFF     6291456  Iomem  R/W    iomem:(iomem_cwt)
 0x4B000000  0x4B0FFFFF     1048576  PCI    R/W    pcimem
 0x60000000  0x619FFFFF    27262976  Local  R/W    main
 0x600088F8  0x61073609    17214738  IText  R/O    main:text
 0x61074000  0x611000FF      573696  IData  R/W    main:data
 0x61100100  0x6128153F     1578048  IBss   R/W    main:bss
 0x61281540  0x619FFFFF     7858880  Local  R/W    main:heap
 0x7B000000  0x7B0FFFFF     1048576  PCI    R/W    pcimem:(pcimem_cwt)
 0x80000000  0x819FFFFF    27262976  Local  R/W    main:(main_k0)
 0xA0000000  0xA19FFFFF    27262976  Local  R/W    main:(main_k1)
```

On the left, the **Start** and **End** addresses correspond to parts of the platform virtual memory map. On the right are the regions and subregions. Subregions are denoted by a name with the **:** separator and no parentheses.

Figure 1-3 illustrates this memory map and its regions.

Figure 1-3 *Memory Map and Regions*



The gaps between the address ranges—for example, **pcimem** ends at **0x4B0FFFFF** and **main** begins at **0x60000000**—are intentional. These gaps allow for expansion of the regions and provide a measure of protection against errant threads. If a runaway thread is advancing through memory writing garbage, it is forced to stop when it hits a gap.

From the output in Example 1-1 and Figure 1-3, you see that the entire DRAM area from 0x60000000 to 0x619FFFFF has been classified as a **local** region and further divided into subregions. These subregions correspond to the various parts of the IOS image itself (text, BSS, and data) and the heap. The heap fills all the local memory area left over after the image is loaded.

Some regions appear to be duplicates, only with a different address range, such as **iomem:(iomem cwt)**:

```
      Start         End      Size(b)  Class  Media  Name
  0x01A00000  0x01FFFFFF     6291456  Iomem  R/W    iomem
  0x31A00000  0x31FFFFFF     6291456  Iomem  R/W    iomem:(iomem_cwt)
 ....
```

These duplicate regions are called *aliases.* Some Cisco platforms have multiple *physical* address ranges that point to the same block of physical memory. These different ranges are used to provide alternate data access methods or automatic data translation in hardware. For example, one address range might provide cached access to an area of physical memory while another might provide uncached access to the same memory.

The duplicate ranges are mapped as alternate views during system initialization and IOS creates alias regions for them. Aliased regions don't count toward the memory total on a platform (because they aren't really separate memory), so they allow IOS to provide a

separate region for alternate memory views without artificially inflating the total memory calculation.

Memory Pools

IOS manages available free memory via a series of *memory pools,* which are essentially heaps in the generic sense; each pool is a collection of memory blocks that can be allocated and deallocated as needed. Memory pools are built out of regions and are managed by the kernel. Often, the pools correspond one-to-one to particular regions, but they are not required to. A memory pool can be built from memory spanning several regions, allowing memory to be allocated and reclaimed from various areas for maximum efficiency.

You can obtain information on IOS memory pools by using the **show memory** command as demonstrated by Example 1-2.

Example 1-2 *IOS Memory Pool Information in* **show memory** *Command Output*

```
router#show memory
               Head   Total(b)    Used(b)    Free(b)  Lowest(b) Largest(b)
Processor  61281540    7858880    3314128    4544752    4377808    4485428
      I/O   1A00000    6291456    1326936    4964520    4951276    4964476
      PCI  4B000000    1048576     407320     641256     641256     641212
...
```

NOTE The output from **show memory** can be very long. It should *not* be issued while console output paging is disabled (the terminal length set to 0) because there is no way to stop or pause the output until it completes.

In Example 1-2, there are three memory pools: **Processor**, **I/O**, and **PCI**. Comparing the **Head** column in this output with the **Start** column from the output of **show region** in Example 1-3 allows you to see which regions are contained within each memory pool.

Example 1-3 **show region** *Command Output*

```
router#show region
Region Manager:
      Start         End     Size(b)  Class  Media  Name
 0x01A00000  0x01FFFFFF     6291456  Iomem  R/W    iomem
 0x31A00000  0x31FFFFFF     6291456  Iomem  R/W    iomem:(iomem_cwt)
 0x4B000000  0x4B0FFFFF     1048576  PCI    R/W    pcimem
 0x60000000  0x619FFFFF    27262976  Local  R/W    main
 0x600088F8  0x61073609    17214738  IText  R/O    main:text
 0x61074000  0x611000FF      573696  IData  R/W    main:data
 0x61100100  0x6128153F     1578048  IBss   R/W    main:bss
 0x61281540  0x619FFFFF     7858880  Local  R/W    main:heap
....
```

From Example 1-3, you can see the Processor memory pool contains memory from the subregion of **main**, called **heap**, which is in class **Local**. The Processor memory pool is common to all IOS systems and always resides in local memory. This is the general memory pool from which data is allocated (such as routing tables).

The **I/O** pool is managing memory from the **iomem** region and the **PCI** pool is managing memory from the **pcimem** region.

The remaining fields in the **show memory** output in Example 1-2 provide useful statistics about the pools as documented in the following list. All units are in bytes.

- **Total**—Total size of the pool.
- **Used**—Current amount of memory allocated.
- **Free**—Current amount of memory available.
- **Lowest**—The least amount of memory ever available since the pool was created.
- **Largest**—The size of the largest contiguous block of memory currently available.

The **show memory** command can also display the blocks within each memory pool, as you'll see later in this chapter.

IOS Processes

IOS processes are essentially equivalent to a single thread in other operating systems—IOS processes have one and only one thread each. Each process has its own stack space, its own CPU context, and can control such resources as memory and a console device (more about that later). To minimize overhead, IOS does not employ virtual memory protection between processes. No memory management is performed during context switches. As a result, although each process receives its own memory allocation, other processes can freely access that same memory.

IOS uses a priority run-to-completion model for executing processes. Initially, it might appear that this non-preemptive model is a poor choice for an operating system that must process incoming packets quickly. In some ways, this is an accurate observation; IOS switching needs quickly outgrew the real-time response limitations of its process model, and in Chapter 2, "Packet Switching Architectures," you'll see how this apparent problem was solved. However, this model still holds some advantages that make it a good fit for support processes that remain outside the critical switching path. Some of these advantages are as follows:

- **Low overhead**—Cooperative multitasking generally results in fewer context switches between threads, reducing the total CPU overhead contributed by scheduling.
- **Less complexity for the programmer**—Because the programmer can control where a process is suspended, it's easy to limit context switches to places where shared data isn't being changed, reducing the possibility for side effects and deadlocks between threads.

Process Life Cycle

Processes can be created and terminated at any time while IOS is operating except during an *interrupt*. A process can be created or terminated by the kernel (during IOS initialization) or by another running process.

NOTE The term *interrupt* used here refers to a hardware interrupt. When the CPU is interrupted, it temporarily suspends the current thread and begins running an interrupt handler function. New processes cannot be created while the CPU is running the interrupt handler.

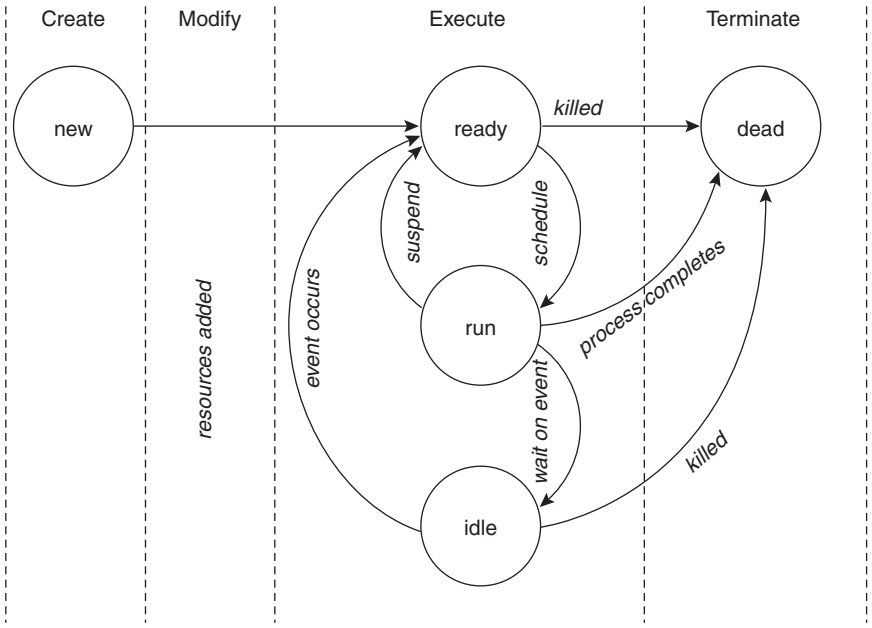
One component in particular is responsible for creating many of the processes in IOS: the *parser*. The parser is a set of functions that interprets IOS configuration and EXEC commands. The parser is invoked by the kernel during IOS initialization and EXEC processes that are providing a command-line interface (CLI) to the console and Telnet sessions.

Any time a command is entered by a user or a configuration line is read from a file, the parser interprets the text and takes immediate action. Some configuration commands result in the setting of a value, such as an IP address, while others turn on complicated functionality, such as routing or event monitoring.

Some commands result in the parser starting a new process. For example, when the configuration command **router eigrp** is entered via the CLI, the parser starts a new process, called *ipigrp* (if the *ipigrp* process hasn't already been started), to begin processing EIGRP IP packets. If the configuration command **no router eigrp** is entered, the parser terminates the ipigrp process and effectively disables any EIGRP IP routing functionality.

IOS processes actually go through several stages during their existence. Figure 1-4 shows these stages and their corresponding states.

Figure 1-4 *Process Life Cycle*



Creation Stage

When a new process is created, it receives its own stack area and enters the *new* state. The process can then move to the modification stage. If no modification is necessary, the process moves to the execution stage.

Modification Stage

Unlike most operating systems, IOS doesn't automatically pass startup parameters or assign a console to a new process when it is created, because it's assumed most processes don't need these resources. If a process does need either of these resources, the thread that created it can *modify* it to add them.

Execution Stage

After a new process is successfully created and modified, it transitions to the *ready* state and enters the execution stage. During this stage, a process can gain access to the CPU and run.

During the execution stage, a process can be in one of three states: ready, run, or idle. A process in the *ready* state is waiting its turn to access the CPU and to begin executing instructions. A process in the *run* state is in control of the CPU and is actively executing instructions. An *idle* process is asleep, waiting on external events to occur before it can be eligible to run.

A process transitions from the ready state to the run state when it's scheduled to run. With non-preemptive multitasking, a scheduled process continues to run on the CPU until it either suspends or terminates. A process can suspend in one of two ways. It can explicitly suspend itself by telling the kernel it wants to relinquish the CPU, transition to the ready state, and wait its next turn to run. A process can also suspend by waiting for an external event to occur. When a process begins waiting on an event, the kernel implicitly suspends it by transitioning it to the idle state, where it remains until the event occurs. After the event occurs, the kernel transitions the process back to the ready state to await its next turn to run.

Termination Stage

The final stage in the process life cycle is the termination stage. A process enters the termination stage when it completes its function and shuts down (called *self termination*) or when another process kills it. When a process is killed or self terminates, the process transitions to the *dead* state. A terminated process remains in the dead state, inactive, until the kernel reclaims all of its resources. The kernel might also record statistics about the process' stack when it terminates. After its resources are reclaimed, the terminated process transitions out of the dead state and is totally removed from the system.

IOS Process Priorities

IOS employs a priority scheme to schedule processes on the CPU. At creation time, every process is assigned one of four priorities based on the process' purpose. The priorities are static; that is, they're assigned when a process is created and never changed. The IOS process priorities are:

- **Critical**—Reserved for essential system processes that resolve resource allocation problems.
- **High**—Assigned to processes that provide a quick response time, such as a process that receives packets directly from a network interface.

- **Medium**—The default priority used by most IOS processes.
- **Low**—Assigned to processes providing periodic background tasks, such as logging messages.

Process priorities provide a mechanism to give some processes preferential access to the CPU based on their relative importance to the entire system. Remember though, IOS doesn't employ preemption, so a higher priority process can't interrupt a lower priority process. Instead, having a higher priority in IOS gives a process more *opportunities* to access the CPU, as you'll see later when you investigate the operation of the kernel's scheduler.

Process Examples

You can use the **show process** command to see a list of all the processes in a system along with some run-time data about each one, as demonstrated in Example 1-4.

Example 1-4 **show process** *Command Output*

```
router#show process
CPU utilization for five seconds: 0%/0%; one minute: 0%; five minutes: 0%
 PID QTy       PC Runtime (ms)    Invoked   uSecs    Stacks TTY Process
   1 M*         0           400         55    727210020/12000  0 Exec
   2 Lst 6024E528       201172      33945    5926 5752/6000   0 Check heaps
   3 Cwe 602355E0            0          1       0 5672/6000   0 Pool Manager
   4 Mst 6027E128            0          2       0 5632/6000   0 Timers
   5 Mwe 602F3E60            0          1       0 5656/6000   0 OIR Handler
   6 Msi 602FA560       290744    1013776     286 5628/6000   0 EnvMon
   7 Lwe 60302944           92      17588       5 5140/6000   0 ARP Input
   8 Mwe 6031C188            0          1       0 5680/6000   0 RARP Input
   9 Mwe 60308FEC        15200     112763     13410748/12000  0 IP Input
  10 Mwe 6033ADC4          420     202811       2 5384/6000   0 TCP Timer
  11 Lwe 6033D1E0            0          1       011644/12000  0 TCP Protocols
  12 Mwe 60389D6C        10204     135198      75 5392/6000   0 CDP Protocol
  13 Mwe 6035BF28        66836    1030665      6411200/12000  0 IP Background
  14 Lsi 60373950            0      16902       0 5748/6000   0 IP Cache Ager
  15 Cwe 6023DD60            0          1       0 5692/6000   0 Critical Bkgnd
  16 Mwe 6023DB80            0         13       0 4656/6000   0 Net Background
  17 Lwe 6027456C            0         11       011512/12000  0 Logger
  18 Msp 6026B0CC          100    1013812       0 5488/6000   0 TTY Background
  19 Msp 6023D8F8            8    1013813       0 5768/6000   0 Per-Second Jobs
  20 Msp 6023D854       405700    1013812     400 4680/6000   0 Net Periodic
  21 Hwe 6023D9BC         5016     101411      49 5672/6000   0 Net Input
  22 Msp 6023D934       135232      16902    8000 5640/6000   0 Per-minute Jobs
```

The following list describes each of the **show process** command output fields found in Example 1-4.

- **PID**—Process identifier. Each process has a unique process identifier number to distinguish it from other processes.
- **Qty**—Process priority and process state. The first character represents the process' priority as follows:
  - **K**—No priority, process has been killed.
  - **D**—No priority, process has crashed.
  - **X**—No priority, process is corrupted.
  - **C**—Critical priority.
  - **H**—High priority.
  - **M**—Medium priority.
  - **L**—Low priority.

  The remaining two characters in this field represent the current state of the process as follows:
  - *****—Process is currently running on the CPU.
  - **E**—Process is waiting for an event (event dismiss).
  - **S**—Process is suspended.
  - **rd**—Process is ready to run.
  - **we**—Process is idle, waiting on an event.
  - **sa**—Process is idle, waiting until a specific absolute time occurs.
  - **si**—Process is idle, waiting for a specific time interval to elapse.
  - **sp**—Process is idle, waiting for a specific time interval to elapse (periodic).
  - **st**—Process is idle, waiting for a timer to expire.
  - **hg**—Process is hung.
  - **xx**—Process is dead.
- **PC**—Contents of the CPU program counter register when the process last relinquished the CPU. This field is a memory address that indicates where the process begins executing the next time it gets the CPU. A value of zero means the process is currently running.
- **Runtime**—Cumulative amount of time (in milliseconds) the process has used the CPU.
- **Invoked**—Total number of times the process has run on the CPU since it was created.

- **uSecs**—Average amount of CPU time (in microseconds) used each time the process is invoked.
- **Stacks**—Stack space usage statistic. The number on the right of the slash (/) shows the total size of the stack space. The number on the left indicates the low water mark for the amount of free stack space available.
- **TTY**—Console device associated with this process. Zero indicates the process does not own a console or communicates with the main system console.
- **Process**—Name of the process. Process names need not be unique (multiple copies of a process can be active simultaneously). However, process IDs are always unique.

If you issue the **show process** command on several different IOS systems, you'll notice some processes appear on every one. Most of these are processes that perform housekeeping or provide services to other processes. Table 1-2 describes the most common of these processes and the tasks they perform.

Table 1-2 *Common System Processes and Their Functions*

System Process	Function
EXEC	Command-line interface (CLI) for the console and directly connected asynchronous TTY lines. The EXEC process accepts user input and provides an interface to the parser.
Pool manager	Manages buffer pools (more on this in the "Packet Buffer Management" section in this chapter).
Check heaps	Periodically validates the integrity of the runtime IOS code and the structure of the memory heap.
Per-minute jobs	Generic system process that runs every 60 seconds performing background maintenance, such as checking the integrity of process stacks.
Per-second jobs	Generic system process that performs tasks that need to be repeated every second.
Critical background	Critical priority process that performs essential system services, such as allocating additional IOS queue elements when they run out.
Net background	Sends interface keepalive packets, unthrottles interfaces, and processes interface state changes.
Logger	Looks for messages (debug, error, and informational) queued via the kernel by other processes and outputs them to the console and, optionally, to a remote syslog server.
TTY background	Monitors directly connected asynchronous TTY lines for activity and starts "EXEC" processes for them when they go active.

All the processes in Table 1-2, except EXEC, are created by the kernel during system initialization and normally persist until IOS is shut down.

IOS Kernel

When used within the context of operating systems, the word *kernel* usually conjures pictures of an operating system core that runs in a special protected CPU mode and manages system resources. Although the IOS kernel does help manage system resources, its architecture differs from those in other operating systems. The IOS kernel is not a single unit but rather a loose collection of components and functions linked with the rest of IOS as a peer rather than a supervisor. There's no special kernel mode. Everything, including the kernel, runs in user mode on the CPU and has full access to system resources.

The kernel schedules processes, manages memory, provides service routines to trap and handle hardware interrupts, maintains timers, and traps software exceptions. The major functions of the kernel are described in more detail in the following sections.

The Scheduler

The actual task of scheduling processes is performed by the *scheduler*. The IOS scheduler manages all the processes in the system using a series of *process queues* that represent each process state. The queues hold context information for processes in that state. Processes transition from one state to another as the scheduler moves their context from one process queue to another. In all, there are six process queues:

- **Idle queue**—Contains processes that are still active but are waiting on an event to occur before they can run.
- **Dead queue**—Contains processes that have terminated but need to have their resources reclaimed before they can be totally removed from the system.
- **Ready queues**—Contain processes that are eligible to run. There are four ready queues, one for each process priority:
  - Critical
  - High
  - Medium
  - Low

When a running process suspends, the scheduler regains control of the CPU and uses an algorithm to select the next process from one of its four ready queues. The steps for this algorithm are as follows:

Step 1 The scheduler first checks for processes waiting in the critical priority ready queue. It runs each critical process, one by one, until all have had a chance to run.

Step 2 After all critical processes have had a chance to run, the scheduler checks the high priority queue. If there are no high priority processes ready, the scheduler skips to Step 3 and checks the medium priority queue.

Otherwise, the scheduler removes each high priority process from the queue and allows it to run. Between each high priority process, the scheduler checks for any critical processes that are ready and runs all of them before proceeding to the next high priority process. After all high priority processes have had their chance, the scheduler skips the medium and low process queues and starts over at Step 1.

Step 3 After there are no high priority processes waiting to run, the scheduler checks the medium priority queue. If there are no medium priority processes ready, the scheduler skips to Step 4 and checks the low priority queue. Otherwise, the scheduler removes each medium priority process from the queue and allows it to run. Between each medium priority process, the scheduler checks for any high priority processes that are ready and runs all of them (interleaved with any critical priority processes) before proceeding to the next medium priority process. After all medium priority processes have had their chance, the scheduler skips over the low priority queue and starts over at Step 1 again. The scheduler skips the low priority queue a maximum of 15 times before proceeding to Step 4. This threshold is a failsafe mechanism to prevent the low priority processes from being starved.

Step 4 After there are no medium or high priority processes waiting to run (or the low priority queue has been skipped 15 times) the scheduler checks the low priority queue. The scheduler removes each low priority process from the queue and allows it to run. Between each low priority process, the scheduler checks for any ready medium priority processes and runs them (interleaved with any high and critical priority processes) before proceeding to the next low priority process.

Step 5 The scheduler finally returns to Step 1 and starts over.

The scheduling algorithm works similar to the operation of a stopwatch. Think of the seconds as representing the critical processes, the minutes as representing the high priority processes, the hours as representing medium processes, and the days as representing the low priority processes. Each pass of the second hand covers all the seconds on the dial. Each pass of the minute hand covers all the minutes on the dial, including a complete pass of all seconds for each minute passed, and so on.

Like a stopwatch, the scheduling algorithm also has a built-in reset feature. The algorithm doesn't advance to check the medium priority queue until there are no high priority processes waiting to run. As long as it finds high priority processes, it starts over from the beginning, checking for critical processes and then high priority processes like a stopwatch that's reset to zero when it reaches the first hour.

Process CPU Utilization

Although IOS does not provide any statistics on the operation of the scheduler itself, there is a way to see how the processes are sharing the CPU. In an earlier example, you saw how the **show process** command gives general statistics about all active processes. A variant of that command, **show process cpu**, has similar output but focuses more on the CPU utilization by each process. Example 1-5 provides some sample output from the **show process cpu** command.

Example 1-5 **show process cpu** *Command Output*

```
router#show process cpu
CPU utilization for five seconds: 90%/82%; one minute: 60%; five minutes: 40%
PID  Runtime(ms)  Invoked  uSecs    5Sec   1Min   5Min TTY Process
   1        1356   2991560      0   0.00%  0.00%  0.00%   0 BGP Router
   2      100804      7374  13670   0.00%  0.02%  0.00%   0 Check heaps
   3           0         1      0   0.00%  0.00%  0.00%   0 Pool Manager
   4           0         2      0   0.00%  0.00%  0.00%   0 Timers
   5        6044         41511000   0.00%  0.00%  0.00%   0 OIR Handler
   6           0         1      0   0.00%  0.00%  0.00%   0 IPC Zone Manager
   7           0         1      0   0.00%  0.00%  0.00%   0 IPC Realm Manager
   8        7700     36331    211   8.00%  0.00%  0.00%   0 IP Input
...
```

The first line of the **show process cpu** output in Example 1-5 shows the overall CPU utilization for the machine averaged over three different time intervals: 5 seconds, 1 minute, and 5 minutes. The 5 second usage statistic is reported as two numbers separated by a slash. The number on the left of the slash represents the total percentage of available CPU capacity used during the last 5-second interval (total percent CPU busy). The number on the right of the slash represents the total percentage of available CPU capacity used while processing interrupts. The 1-minute usage and the 5-minute usage show the total percent CPU usage as an exponentially decaying average over the last minute and the last 5 minutes, respectively.

Below the general usage statistics, the same three intervals are reported on a per process basis. Note, the scheduler automatically computes all these usage statistics every 5 seconds and stores them internally in case someone issues a **show process cpu** command. So, it's possible to see the exact same statistics twice—not updated—if the command is issued twice within a 5 second period.

Using the **show process cpu** output, it's fairly easy to determine what processes are most active and how much of the CPU capacity they use. In the example, an average of 90 percent of the CPU capacity was being used during the 5-second interval right before the command was issued. An average of 82 percent of capacity was being used by interrupt processing with the other 8 percent being used by scheduled processes (90% – 82% = 8%). Over the last 60 seconds, an average of 60 percent of the CPU capacity was being used, and over the last 5 minutes, 40 percent of the CPU capacity was being used, on average. Looking at the

individual scheduled processes, 8 percent of the CPU capacity was being consumed by the **ip_input** process during the last 5 seconds. This accounts for all the process usage; so, the other processes either weren't scheduled on the CPU during that interval or they used less than 0.01 percent of the CPU.

In this example, a relatively large percentage of the CPU capacity is being consumed by interrupt processing. So what is IOS doing during that 82 percent of the time? Unfortunately, there's no detailed account of interrupt processing CPU usage like there is for processes; so, we really can't pinpoint the usage to a specific source. However, we can make a good estimate about how this CPU time is being used. As you'll see in Chapter 2, when Fast Switching is being used for packets, most of the packet switching work is performed during an interrupt. IOS does do some other processing during interrupts, but by far most of the processing time is used for fast packet switching. A high interrupt CPU usage statistic usually just means IOS is fast switching a large volume of packets.

The remaining unused CPU capacity (10 percent in our example) is called *idle* CPU time. "Idle," though, is a somewhat misleading term because a CPU is never really idle while it's running; it's always executing instructions for something. In IOS, that idle time actually represents the time the CPU is spending running the scheduler itself—a very important task indeed. It's important for an IOS system to operate with sufficient idle time. As CPU utilization approaches 100 percent, very little time is left over for the scheduler, creating a potentially serious situation. Because the scheduler is responsible for running all the background critical support processes, if it can't run, neither can the processes and that can lead to system failure.

As you'll see, IOS has watchdog timers to prevent runaway processes from consuming all the CPU capacity. Unfortunately, IOS does not always employ this type of limiting mechanism for interrupts. Although many platforms do support limiting CPU interrupt time through *interrupt throttling*, it's not always enabled by default.

In any event, watchdog timers and interrupt throttling are just safety mechanisms. When planning for a system that might encounter a high rate of packets (many busy high speed network interfaces, for example), it's best to select a sufficiently fast CPU and configure IOS with the most efficient switching methods to avoid CPU starvation problems.

Watchdog Timer

To reduce the impact of runaway processes, IOS employs a process *watchdog* timer to allow the scheduler to periodically poll the current running process. This feature is not the same as preemption but is instead a failsafe mechanism to prevent the system from becoming unresponsive or completely locking up due to a process consuming all of the CPU. If a process appears to be hung (for example, it's been running for a long time), the scheduler can force the process to terminate.

Each time the scheduler allows a process to run on the CPU, it starts a watchdog timer for that process. After a preset period, two seconds by default, if the process is still running, the timer expires and the scheduler regains control. The first time the watchdog expires, the scheduler prints a warning message for the process similar to the following, and then continues the process on the CPU:

```
%SNMP-3-CPUHOG: Processing GetNext of ifEntry.17.6
%SYS-3-CPUHOG: Task ran for 2004 msec (49/46), Process = IP SNMP, PC = 6018EC2C
 -Traceback= 6018EC34 60288548 6017E9C8 6017E9B4
```

If the watchdog expires a second time and the process still hasn't suspended, the scheduler terminates the process.

The Memory Manager

The kernel's memory manager is responsible, at a macro level, for managing all the memory available to IOS, including the memory containing the IOS image itself. The memory manager is actually not one single component but three separate components, each with its own responsibility:

- **Region Manager**—Defines and maintains the various memory regions on a platform.
- **Pool Manager**—Manages the creation of memory pools and the allocation/de-allocation of memory blocks within the pools.
- **Chunk Manager**—Manages the specially allocated memory blocks that contain multiple fixed-sized sub-blocks.

Region Manager

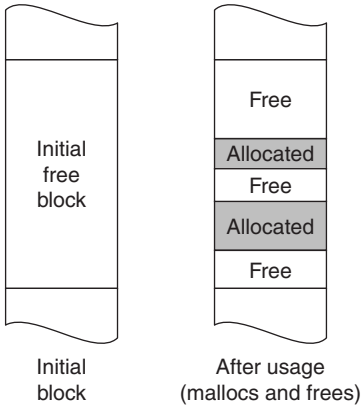
The region manager is responsible for maintaining all the memory regions. It provides services allowing other parts of IOS to create regions and to set their attributes. It also allows others to query the available memory regions, for example, to determine the total amount of memory available on a platform.

Pool Manager

The memory pool manager is a very important component of the kernel. Just as the scheduler manages allocating CPU resources to processes, the pool manager manages memory allocation for processes. All processes must go through the pool manager, directly or indirectly, to allocate memory for their use. The pool manager is invoked each time a process uses the standard system functions *malloc* and *free* to allocate or return memory.

The pool manager operates by maintaining lists of free memory blocks within each pool. Initially, of course, each pool contains just one large free block equal to the size of the pool. As the pool manager services requests for memory, the initial free block gets smaller and smaller. At the same time, processes can release memory back to the pool, creating a number of discontiguous free blocks of varying sizes. This scenario is called *memory fragmentation* and is illustrated in Figure 1-5.

Figure 1-5 *Memory Pool Allocation*

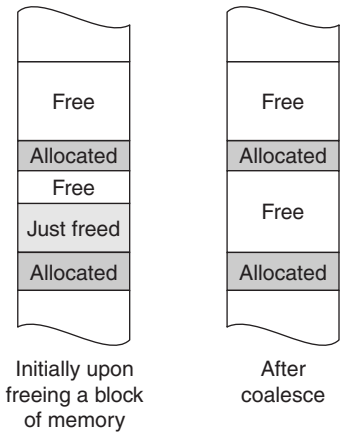


As free blocks are returned to a pool, the pool manager adds their size and starting addresses to one of the pool's free lists for blocks of similar size. By default, the pool manager maintains free lists for each of the following block sizes: 24, 84, 144, 204, 264, 324, 384, 444, 1500, 2000, 3000, 5000, 10,000, 20,000, 32,768, 65,536, 131,072, and 262,144 bytes. Note that these sizes have no relation to the system buffers discussed later in this chapter.

When a process requests memory from a pool, the pool manager first starts with the free list associated with the size of the request. This method helps make efficient use of memory by matching requests to recycled blocks that are close in size. If there are no blocks available on the best-fit list, the pool manager continues with each higher list until a free block is found. If a block is found on one of the higher lists, the pool manager splits the block and puts the unused portion back on the appropriate list of smaller blocks.

The pool manager tries to control fragmentation by coalescing returned blocks that are physically adjacent to one another. When a block is returned and it's next to an existing free block, the pool manager combines the two blocks into one larger block and places the resulting block on the appropriate sized list, as shown in Figure 1-6.

Figure 1-6 *Free Block Coalesce*



Earlier, Example 1-2 demonstrated how the EXEC command **show memory** displays names and statistics for the available memory pools in its summary. The **show memory** command also lists detail about the individual blocks contained in each of the pools. Following the pool summary, the command output lists all the blocks, in order, under each pool, as demonstrated in the output in Example 1-6.

Example 1-6 **show memory** *Command Output with Memory Pool Details*

```
router#show memory
              Head   Total(b)    Used(b)    Free(b)  Lowest(b) Largest(b)
Processor  61281540    7858880    3314128    4544752    4377808    4485428
      I/O   1A00000    6291456    1326936    4964520    4951276    4964476
      PCI  4B000000    1048576     407320     641256     641256     641212

Processor memory

 Address  Bytes Prev.    Next     Ref  PrevF   NextF   Alloc PC  What
61281540   1460 0        61281B20   1                  6035CCB0  List Elements
61281B20   2960 61281540 612826DC   1                  6035CCDC  List Headers
612826DC   9000 61281B20 61284A30   1                  60367224  Interrupt Stack
61284A30     44 612826DC 61284A88   1                  60C8BEEC  *Init*
61284A88   9000 61284A30 61286DDC   1                  60367224  Interrupt Stack
61286DDC     44 61284A88 61286E34   1                  60C8BEEC  *Init*
61286E34   9000 61286DDC 61289188   1                  60367224  Interrupt Stack
61289188     44 61286E34 612891E0   1                  60C8BEEC  *Init*
612891E0   4016 61289188 6128A1BC   1                  602F82CC  TTY data
6128A1BC   2000 612891E0 6128A9B8   1                  602FB7B4  TTY Input Buf
6128A9B8    512 6128A1BC 6128ABE4   1                  602FB7E8  TTY Output Buf
```

The memory pool detail fields displayed in Example 1-6 are as follows:

- **Address**—Starting address of the block.
- **Bytes**—Size of the block.
- **Prev**—Address of the preceding block in the pool.
- **Next**—Address of the following block in the pool.
- **Ref**—Number of owners of this memory block.
- **PrevF**—For free blocks only, the address of the preceding block in the free list.
- **NextF**—For free blocks only, the address of the next block in the free list.
- **Alloc PC**—Value of the CPU's program counter register when the block was allocated. This value can be used to determine which process allocated the block.
- **What**—Description of how the block is being used.

You can use a variation of this command, **show memory free**, to display the free blocks in each pool, as demonstrated in Example 1-7. The command output lists each free block in order by free list. Empty free lists appear with no memory blocks listed under them.

Example 1-7 **show memory free** *Command Output*

```
router#show memory free
...

Processor memory

 Address  Bytes Prev.    Next     Ref  PrevF   NextF   Alloc PC  What
             24    Free list 1
6153E628     52 6153E5F0 6153E688   0  0       615A644 603075D8  Exec
615A6444     44 615A63F8 615A649C   0  6153E62 0       603567F0  (fragment)
             92    Free list 2
            112    Free list 3
            116    Free list 4
            128    Free list 5
61552788    128 6155273C 61552834   0  0       0       603075D8  (coalesced)
            132    Free list 6
            152    Free list 7
            160    Free list 8
            208    Free list 9
            224    Free list 10
            228    Free list 11
6153E460    240 6153E428 6153E57C   0  0       0       6047A960  CDP Protocol
            272    Free list 12
            288    Free list 13
            296    Free list 14
            364    Free list 15
            432    Free list 16
            488    Free list 17
            500    Free list 18
```

continues

Example 1-7 show memory free *Command Output (Continued)*

```
6153E6D4    544 6153E69C 6153E920   0  0       0       60362350  (coalesced)
           1500    Free list 19
           2000    Free list 20
61552B84   2316 61552954 615534BC   0  0       0       603075D8  (coalesced)
           3000    Free list 21
6153EA14   4960 6153E9D4 6153FDA0   0  0       0       603080BC  (coalesced)
           5000    Free list 22
          10000    Free list 23
61572824  15456 6157106C 615764B0   0  0       0       603080BC  (coalesced)
          20000    Free list 24
          32768    Free list 25
6159BA64  35876 61598B6C 615A46B4   0  0       0       60371508  (fragment)
          65536    Free list 26
         131072    Free list 27
         262144    Free list 28

Total: >   59616
```

Chunk Manager

The pool manager provides a very effective way for managing a collection of random-sized blocks of memory. However, this feature does not come without cost. The pool manager introduces 32 bytes of memory overhead on every block managed. Although this overhead is insignificant for pools with a few hundred large blocks, for a pool with thousands of small blocks, the overhead can quickly become substantial. As an alternative, the kernel provides another memory manager, called the *chunk manager*, that can manage large pools of small blocks without the associated overhead.

Unlike the pool manager, the chunk manager does not create additional memory pools with free lists of varying size. Instead, the chunk manager manages a set of fixed-size blocks subdivided within a larger block that has been allocated from one of the standard memory pools. In some ways, the chunk manager can almost be considered a submemory pool manager.

The strategy most often employed is this: A process requests the allocation of a large memory block from a particular memory pool. The process then calls on the chunk manager to subdivide the block into a series of smaller fixed-size chunks and uses the chunk manager to allocate and free the chunks as needed. The advantage is there are only 32 bytes of overhead (associated with the one large block) and the pool manager is not burdened with allocating and reclaiming thousands of little fragments. So, the potential for memory fragmentation in the pool is greatly reduced.

Process Memory Utilization

IOS has another variation of the **show process** CLI command, **show process memory**, that shows the memory utilization by each process. This command allows you to quickly determine how available memory is being allocated by processes in the system. Example 1-8 provides some sample output from a **show process memory** command.

Example 1-8 **show process memory** *Command Output*

```
router#show process memory
Total: 7858880, Used: 3314424, Free: 4544456
 PID TTY  Allocated      Freed    Holding    Getbufs    Retbufs Process
   0   0      86900       1808    2631752          0          0 *Init*
   0   0        448      55928        448          0          0 *Sched*
   0   0    7633024    2815568       6348     182648          0 *Dead*
   1   0        268        268       3796          0          0 Load Meter
   2   0        228          0       7024          0          0 CEF Scanner
   3   0          0          0       6796          0          0 Check heaps
   4   0         96          0       6892          0          0 Pool Manager
   5   0        268        268       6796          0          0 Timers
...

  65   0          0      14492       6796          0      13248 Per-minute Jobs
  66   0     143740       3740     142508          0          0 CEF process
  67   0          0          0       6796          0          0 xcpa-driver
                                  3314184 Total
```

The first line in Example 1-8 shows the total number of bytes in the pool followed by the amount used and the amount free. The statistics in the first line represent metrics for the Processor memory pool and should match the output of the **show memory** command. The summary line is followed by detail lines for each scheduled process:

- **PID**—Process identifier.
- **TTY**—Console assigned to the process.
- **Allocated**—The total number of bytes this process has allocated since it was created.
- **Freed**—The total number of bytes this process has freed since it was created.
- **Holding**—The number of bytes this process currently has allocated. Note that, because a process can free memory that was allocated by another process, the **allocated**, **freed**, and **holding** numbers might not always balance. In other words, **allocated** minus **freed** does not always equal **holding.**
- **Getbufs**—The total number of bytes for new packet buffers created on behalf of this process. Packet buffer pools are described later in the "Packet Buffer Management" section.

- **Retbufs**—The total number of bytes for packet buffers trimmed on behalf of this process. Packet buffer trims are described later in the "Packet Buffer Management" section.
- **Process**—The name of the process.

Notice there are three processes appearing in the **show process memory** output labeled ***Init***, ***Sched***, and ***Dead***. These are not real processes at all but are actually summary lines showing memory allocated by IOS outside the scheduled processes. The following list describes these summary lines:

- ***Init***—Shows memory allocated for the kernel during system initialization before any processes are created.
- ***Sched***—Shows memory allocated by the scheduler.
- ***Dead***—Shows memory once allocated by scheduled processes that has now entered the dead state. Memory allocated to dead processes is eventually reclaimed by the kernel and returned to the memory pool.

Memory Allocation Problems

In all the examples thus far, we've assumed that when a process requests an allocation of memory its request is granted. However, this might not always be the case. Memory resources, like CPU resources, are limited; there's always the chance that there might not be enough memory to satisfy a request. What happens when a process calls on the pool manager for a memory block and the request fails? Depending on the severity (that is, what the memory is needed for), the process might continue to limp along or it might throw up its hands and quit. In either case, when the pool manager receives a request that it can't satisfy, it prints a warning message to the console similar to the following:

```
%SYS-2-MALLOCFAIL: Memory allocation of 2129940 bytes failed from 0x6024C00C,
  pool I/O, alignment 32
 -Process= "OIR Handler", ipl= 4, pid= 7
 -Traceback= 6024E738 6024FDA0 6024C014 602329C8 60232A40 6025A6C8 60CEA748
  60CDD700 60CDF5F8 60CDF6E8 60CDCB40 60286F1C 602951
```

Basically, there are two reasons why a valid request for memory allocation can fail:

- There isn't enough free memory available.
- Enough memory is available, but it's fragmented so that no contiguous block is available of sufficient size.

Failures due to lack of memory usually occur because there is simply not enough memory on the platform to support all the activities in the system. When this happens, you have two choices: add more memory to the platform, or reduce configured features, interfaces, and so on until the problem goes away. In some rare cases, though, these "out of memory" failures can occur because of a defect called a *memory leak* in one of the processes. The term *memory leak* refers to a scenario where a process continually allocates memory blocks

but never frees any of them; eventually it consumes all available memory. Memory leaks usually are caused by software defects.

Allocation failures also can occur when there's seemingly plenty of memory available. For an example, look at the following sample **show memory** output in Example 1-9.

Example 1-9 *Sample* **show memory** *Command Output*

```
router#show memory
               Head   Total(b)    Used(b)    Free(b)  Lowest(b) Largest(b)
Processor  61281540    7858880    4898452    2960428      10507       8426
....
```

This particular system has 2,960,428 bytes of memory free. Yet, what happens when a process tries to allocate just 9000 bytes? The request fails. The clue to why is in the **Largest** field of the **show memory** output. At the time this command was issued, the largest contiguous block of memory available was only 8426 bytes long—not large enough to satisfy a 9000 byte request.

This scenario is indicative of severe memory fragmentation. Severe fragmentation occurs when many small memory blocks are allocated and then returned in such a way that the pool manager can't coalesce them into larger blocks. As this trend continues, the large contiguous blocks in the pool are chipped away until nothing is left but smaller fragments. When all the larger blocks are gone, memory allocation failures start to occur.

The kernel's pool manager is designed to maintain self-healing memory pools that avoid fragmentation, and in most cases this design works without a problem. However, some allocation scenarios can break this design; one example being the pool of many small blocks discussed earlier.

Packet Buffer Management

In packet routing, as in any store-and-forward operation, there must be some place within the system to store the data while it is being routed on its way. The typical strategy is to create memory buffers to hold incoming packets while the switching operation is taking place. Because the capability to route packets is central to the IOS architecture, IOS contains a special component dedicated to managing just such buffers. This component is called the *buffer pool manager* (not to be confused with the memory pool manager discussed earlier). IOS uses this component to create and manage a consistent series of packet buffer pools for switching on every platform. The buffers in these pools are collectively known as the *system buffers*.

The buffer pool manager provides a convenient way to manage a set (or *pool*) of buffers of a particular size. Although it can be used to manage any type of buffer pool, the buffer pool manager is used primarily to manage pools of packet buffers, so that's what we'll focus on here.

Packet buffer pools are created using memory allocated from one of the available memory pools: for example, Processor, I/O, and so on. To create a pool, the packet buffer manager requests a block of memory from the kernel's memory pool manager and divides it into buffers. The packet buffer manager then builds a list of all the free buffers and tracks the allocation and freeing of those buffers.

Packet buffer pools can be either static or dynamic. Static pools are created with a fixed number of buffers—no additional buffers can be added to the pool. Dynamic pools are created with a particular base minimum number of buffers, called *permanent* buffers, but additional buffers can be added or removed on demand. With dynamic buffer pools, if the buffer pool manager receives a request while the pool is empty, it attempts to expand the pool and satisfy the request immediately. If it's not possible to expand the pool within the context of the request, it fails the request and expands the pool later in the **pool manager** background process.

Packet buffer pools are classified as either public or private. Public pools, as the name implies, are available for use by any system process, and private pools are created for (and known only to) a specific subset of processes that use them.

System Buffers

Every IOS system has a predefined set of public buffer pools known as the *system buffers*. These buffer pools are used for process switching packets through the platform and for creating packets that originate in the platform (such as interface keepalive packets and routing updates). IOS provides statistics about these and other buffer pools through the **show buffer** command. For an example, let's examine the **show buffer** output in Example 1-10.

Example 1-10 show buffer *Command Output*

```
router#>show buffer
Buffer elements:
     500 in free list (500 max allowed)
     747314 hits, 0 misses, 0 created

Public buffer pools:
Small buffers, 104 bytes (total 50, permanent 50):
     46 in free list (20 min, 150 max allowed)
     530303 hits, 6 misses, 18 trims, 18 created
     0 failures (0 no memory)
Middle buffers, 600 bytes (total 25, permanent 25):
     25 in free list (10 min, 150 max allowed)
     132918 hits, 3 misses, 9 trims, 9 created
     0 failures (0 no memory)
```

Example 1-10 show buffer *Command Output (Continued)*

```
Big buffers, 1524 bytes (total 50, permanent 50):
     50 in free list (5 min, 150 max allowed)
     47 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
VeryBig buffers, 4520 bytes (total 10, permanent 10):
     10 in free list (0 min, 100 max allowed)
     26499 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
Large buffers, 5024 bytes (total 0, permanent 0):
     0 in free list (0 min, 10 max allowed)
     0 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
Huge buffers, 18024 bytes (total 0, permanent 0):
     0 in free list (0 min, 4 max allowed)
     0 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
...
```

Notice the list of **Public buffer pools**. These are the standard system buffers for IOS. Each one has a name, such as **Small buffers**, **Middle buffers**, and so on, that identifies the particular pool. Following the pool name is the byte size of the buffers contained in that pool. Each buffer pool contains buffers of one and only one size. The sizes vary from 104 bytes up to 18,024 bytes in order to accommodate the various maximum transmission unit (MTU) sizes encountered on different interface media.

The remainder of the fields in Example 1-10 are as follows:

- **total**—The total number of buffers in the pool (both allocated and free).
- **permanent**—The base number of buffers in the pool. For dynamic pools, the total number of buffers can fluctuate as the pool grows and shrinks. However, there will never be less than the **permanent** number of buffers in the pool.
- **in free list**—Indicates the number of free buffers available.
- **min**—The minimum number of buffers *desired* in the free list. If the number of free buffers drops below **min**, the buffer pool manager attempts to grow the pool and add more buffers until the number of free buffers is greater than **min**.
- **max allowed**—The maximum number of buffers *desired* in the free list. If the number of free buffers increases beyond **max allowed**, the buffer pool manager attempts to shrink the pool, deleting free buffers, until the number of free buffers drops below **max allowed**. Note that the buffer pool manager never shrinks the size of the pool below the **permanent** metric regardless of the value of **max allowed**.
- **hits**—The number of buffers that have been requested from the pool.
- **misses**—The number of times a buffer was requested from the pool when there were fewer than **min** buffers in the free list.

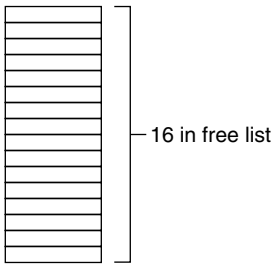
- **trims**—The number of buffers that have been deleted from the pool during pool shrinkage.
- **created**—The number of buffers that have been created and added to the pool during pool growth.
- **failures**—The number of times a buffer was requested from the pool but the request could not be satisfied. Failures can occur for two reasons:
  - — A buffer request occurs during an interrupt and there are no free buffers. Pools can't grow during an interrupt.
  - — A buffer request occurs, there are no free buffers, and there is no more memory left in the memory pool to grow the buffer pool.
- **no memory**—The number of times a failure occurred because there was no available memory in the memory pool. Note that this counter is no longer used in recent versions of IOS.

To get an understanding of just how the buffer pool manager operates with these system buffers, let's step through a packet-switching scenario and monitor how the **show buffers** counters change for a particular pool. Consider the following example.

Let's start with 16 buffers in the free list, as shown in the following output and illustrated in Figure 1-7.

```
Small buffers, 104 bytes (total 16, permanent 16):
     16 in free list (8 min, 16 max allowed)
     0 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
```

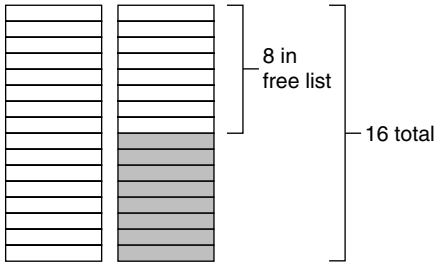
Figure 1-7 *Beginning with an Empty Buffer*



Now say IOS receives eight packets that fit nicely into the 104-byte small buffers. The packet receive software requests and receives eight buffers from the small buffer pool and copies in the packets. The free list now drops from 16 to 8, resulting in a count of eight free buffers and eight hits, as illustrated in the following output and in Figure 1-8.

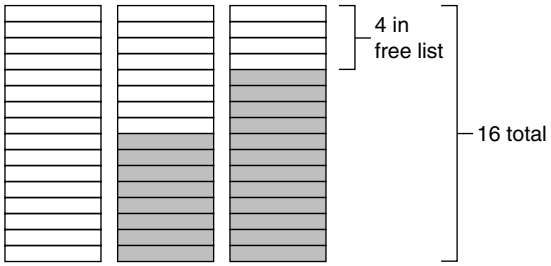
```
Small buffers, 104 bytes (total 16, permanent 16):
     8 in free list (8 min, 16 max allowed)
     8 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
```

Figure 1-8 *Eight Packets Received*



Now IOS receives four more packets before processing any of the eight previously received. The packet receive software requests four more buffers from the **small** pool and copies in the data packets. Figure 1-9 illustrates the pool at this point.

Figure 1-9 *Four More Packets Received*



Looking at the output of **show buffers**, we see several counters have incremented. Let's inspect them to see what happened to the pool:

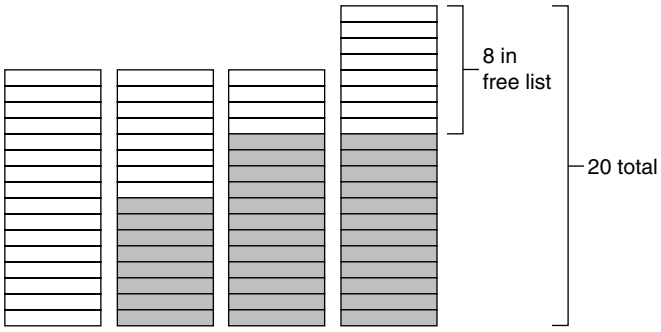
```
Small buffers, 104 bytes (total 16, permanent 16):
     4 in free list (8 min, 16 max allowed)
     12 hits, 4 misses, 0 trims, 0 created
     0 failures (0 no memory)
```

The number in the free list has now dropped to 4 and there are now 12 hits, reflecting the 4 new buffers requested. Why have the misses incremented to four, though? Because each time a buffer is requested from a pool where it causes the free list to drop below the minimum free buffers allowed, a miss is counted. We can see from the output that the minimum free is eight and we now have only four buffers in the free list; so, IOS counted four misses.

Because there are now fewer than **min** buffers in the free list, the buffer pool manager runs the pool manager process to grow the pool and to create more free buffers, as demonstrated in the following output and illustrated in Figure 1-10.

```
Small buffers, 104 bytes (total 20, permanent 16):
     8 in free list (8 min, 16 max allowed)
     12 hits, 4 misses, 0 trims, 4 created
     0 failures (0 no memory)
```

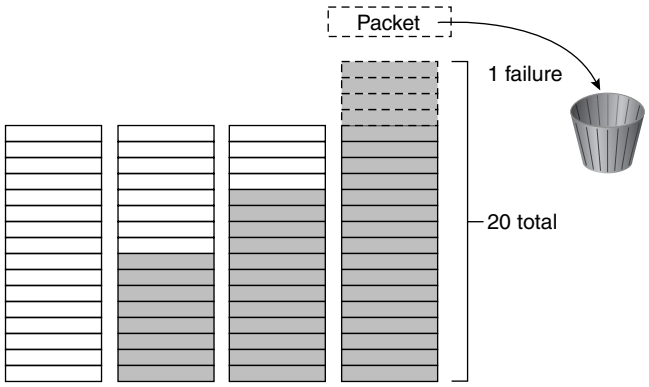
Figure 1-10 *After Pool Manager Creates New Buffers*



We now see that the pool manager process has created four new buffers to bring the number in the free list back up to a total of eight, the minimum number of buffers desired to be free at any time in this pool.

Now, Figure 1-11 illustrates what happens when IOS tries to request nine more packet buffers from this pool before returning any free buffers.

Figure 1-11 *Storing Nine More Packets*

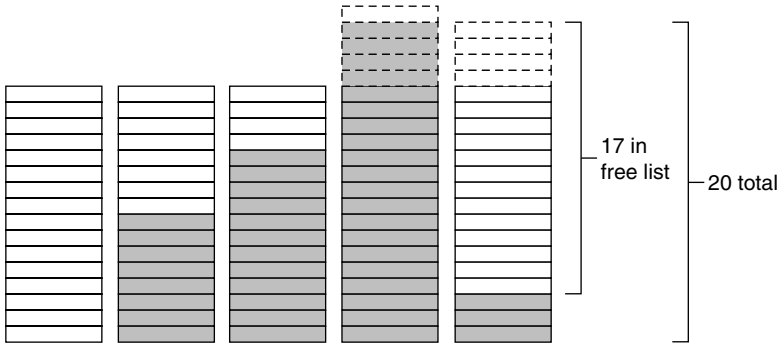


From the following output, we see that the free list has been completely exhausted—0 buffers remaining—and there have been 20 hits, 13 misses, and 1 failure. The failure accounts for the one request that overran the buffer pool.

```
Small buffers, 104 bytes (total 20, permanent 16):
     0 in free list (8 min, 16 max allowed)
     20 hits, 13 misses, 0 trims, 4 created
     1 failures (0 no memory)
```

Finally, IOS begins processing packets and returning buffers to the pool. IOS processes 17 of the 20 packets and returns the 17 free buffers, as illustrated in Figure 1-12 and the output that follows.

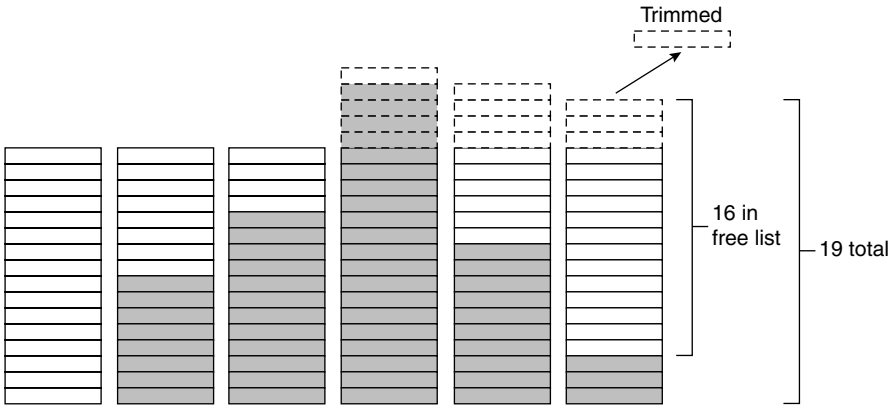
Figure 1-12 *Returning 17 Buffers to the Pool*




```
Small buffers, 104 bytes (total 20, permanent 16):
     17 in free list (8 min, 16 max allowed)
     20 hits, 13 misses, 0 trims, 4 created
     1 failures (0 no memory)
```

The **small** pool now has 17 in the free list. However, the desired maximum free buffers is only 16. Because the number in the free list exceeds the **max allowed**, the next time the **pool manager** process runs it shrinks the buffer pool to adjust the number of free buffers, causing one to be trimmed, as shown in Figure 1-13 and the output that follows.

Figure 1-13 *Trimming the Extra Buffer*



```
Small buffers, 104 bytes (total 20, permanent 16):
     17 in free list (8 min, 16 max allowed)
     20 hits, 13 misses, 1 trims, 4 created
     1 failures (0 no memory)
```

Finally, IOS processes all the remaining packets and returns the buffers to the pool. Eventually, the pool manager process trims all the remaining buffers above the permanent pool size, resulting in the following counts:

```
Small buffers, 104 bytes (total 16, permanent 16):
     16 in free list (8 min, 16 max allowed)
     20 hits, 13 misses, 4 trims, 4 created
     1 failures (0 no memory)
```

Device Drivers

One of the primary functions of an operating system is to provide hardware abstraction between platform hardware and the programs that run on it. This hardware abstraction typically occurs in device drivers that are part of the operating system, making those components an integral part of the system. IOS is no different in that regard. IOS contains device drivers for a range of platform hardware devices, such as Flash cards and NVRAM, but most notable are its device drivers for network interfaces.

IOS network interface device drivers provide the primary intelligence for packet operations in and out of interfaces. Each driver consists of two main components: a control component and a data component. The control component is responsible for managing the state and status of the device (for example, shutting down an interface). The data component is responsible for all data flow operations through the device and contains logic that assists with the packet switching operations. IOS device drivers are very tightly coupled with the packet switching functions, as we discuss in Chapter 2.

IOS device drivers interface with the rest of the IOS system via a special control structure called the *interface descriptor block* (IDB). The IDB contains, among other things, entry points into a device driver's functions and data about a device's state and status. For example, the IP address, the interface state, and the packet statistics are some of the fields that are present in the IDB. IOS maintains an IDB for each interface present on a platform and maintains an IDB for each subinterface.

Summary

Cisco IOS began as a small embedded system and has been expanded over time to become the powerful network operating system it is today. The basic components for IOS are no different than the ones used to build other operating systems. However, in IOS those components are optimized to work within the environment in which IOS is used: limited memory and speed-critical packet switching.

IOS is a cooperative multitasking operating system and operates within a single flat address space. All program code, buffers, routing tables, and other data, reside in the same address space, and all processes can address each other's memory. IOS processes are equivalent to threads in other operating systems.

IOS has a small kernel that provides CPU scheduling services and memory management services to processes. Unlike other operating systems, the IOS kernel runs in "user" mode on the CPU (peer-to-processes) and shares the same address space with the rest of the system.

IOS device drivers provide a hardware abstraction layer between the media hardware and the operating system. The network interface device drivers are tightly coupled to the packet switching software. In Chapter 2, "Packet Switching Architecture," we examine this packet switching software in more detail.