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

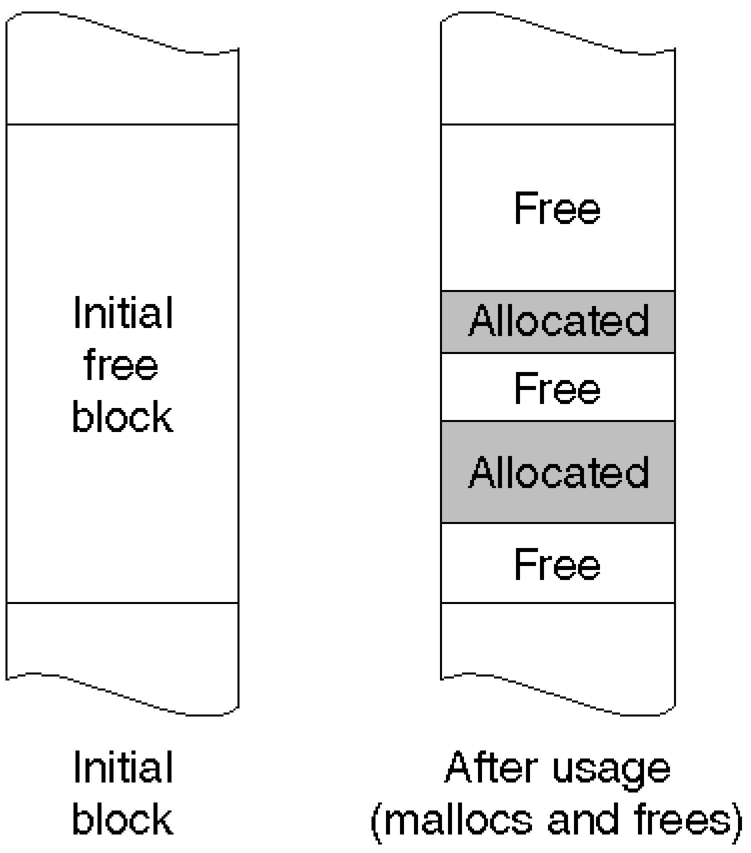
The region manager is responsible for maintaining all the memory regions. It provides services allowing other parts of IOS to create regions and to set their attributes. It also allows others to query the available memory regions, for example, to determine the total amount of memory available on a platform.

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

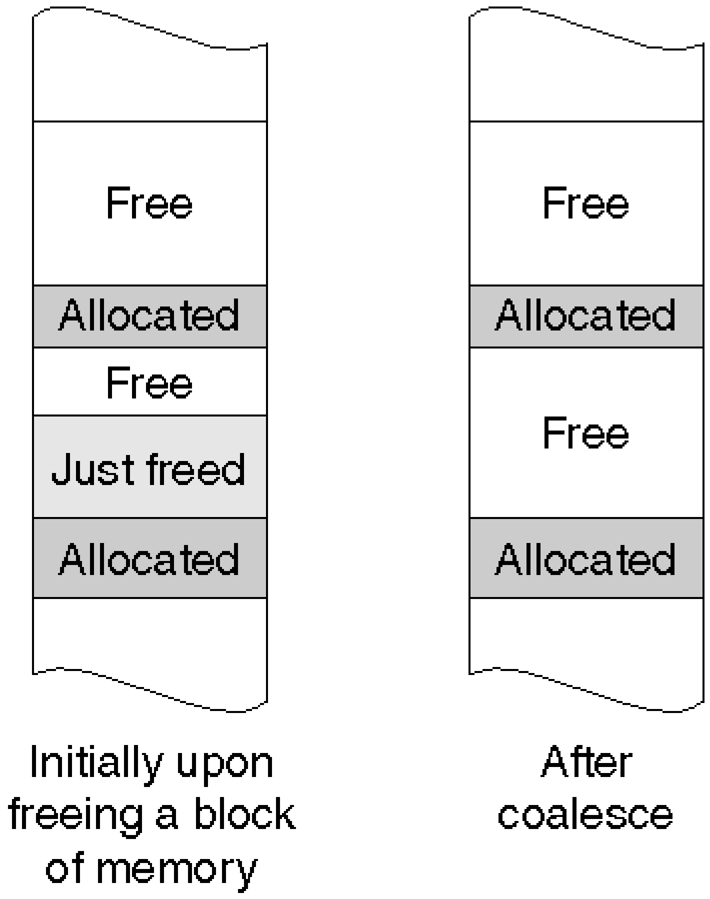


As free blocks are returned to a pool, the pool manager adds their size and starting addresses to one of the pool's free lists for blocks of similar size. By default, the pool manager maintains free lists for each of the following block sizes: 24, 84, 144, 204, 264, 324, 384, 444, 1500, 2000, 3000, 5000, 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 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.

< BACK Make Note | Bookmark CONTINUE >

Index terms contained in this section