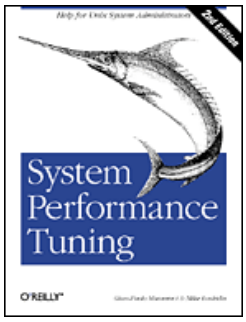

System Performance Tuning, 2nd Edition

By Gian-Paolo D. Musumeci and Mike Loukides
2nd Edition February 2002
0-596-00284-X, Order Number: 284X
336 pages, $39.95 US $59.95 CA

Chapter 4
Memory

Ideally one would desire an indefinitely large memory capacity such that any particular. . . word would be immediately available. . . .
A. W. Burks, H. H. Goldstine, and J. von Neumann: Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946

In this chapter:
Implementations of Physical Memory
Virtual Memory Architecture
Paging and Swapping
Consumers of Memory
Tools for Memory Performance Analysis
Concluding Thoughts

Physical memory is a set of integrated circuits designed to store binary data. This storage has two characteristic properties: it is *transient*, as all stored information vanishes when electrical power is lost, and *randomly accessible*, meaning any bit can be accessed as fast as any other bit. In addition to physical memory, most systems implement virtual memory, which acts to manage physical memory and provide a simple interface to application developers. Virtual memory is consumed by the system kernel, filesystem caches, intimately shared memory, and processes.

Memory performance begins to affect overall system performance in two instances. The first instance occurs when the system is unable to retrieve and store data from physical memory fast enough, or when the system is forced to travel to main memory frequently. This sort of problem can be attacked by tuning the algorithm that is responsible or by buying a system with faster access to main memory. The second, and more likely, case is that the demand for physical memory by all currently running applications, including the kernel, exceeds the available amount. The system is then forced to begin *paging*, or writing unused pieces of memory to disk. If the low memory condition worsens, the memory consumed by entire processes will be written to disk, which is called *swapping*.[1] Memory conditions fall into four categories:

- Sufficient memory is available, and the system performs optimally.
- Memory is constrained (one likely culprit, especially on older Solaris systems, is the filesystem cache). Performance begins to suffer as the system attempts to scavenge memory that is not in active use.
- The system is legitimately short of memory. Performance suffers, especially on interactive processes.
- Memory is critically scarce. Swapping begins to take place. System performance takes a heavy hit, and interactive performance becomes abysmal.

In this chapter, I'll describe how memory is physically implemented, the mechanics of how the system manages memory, and how paging and swapping function at a system level. I'll give you some tools you can use to analyze your memory use, and explain how to work with swap space. Finally, I'll cover mechanisms to address some common memory performance problems.

Implementations of Physical Memory

Let's start by looking at how memory is physically implemented in modern systems. All modern, fast memory implementations are accomplished via semiconductors,[2] of which there are two major types: *dynamic random access memory* (DRAM) and *static random access memory* (SRAM). The difference between them is how each memory cell is designed. Dynamic cells are charge-based, where each bit is represented by a charge stored in a tiny capacitor. The charge leaks away in a short period of time, so the memory must be continually refreshed to prevent data loss. The act of reading a bit also serves to drain the capacitor, so it's not possible to read that bit again until it has been refreshed. Static cells, however, are based on gates, and each bit is stored in four or six connected transistors. SRAM memories retain data as long as they have power; refreshing is not required. In general, DRAM is substantially cheaper and offers the highest densities of cells per chip; it is smaller, less power-intensive, and runs cooler. However, SRAM is as much as an order of magnitude faster, and therefore is used in high-performance environments. Interestingly, the Cray-1S supercomputer had a main memory constructed entirely from SRAM. The heat generated by the memory subsystem was the primary reason that system was liquid-cooled.

There are two primary performance specifications for memory. The first represents the amount of time required to read or write a given location in memory, and is called the *memory access time*. The second, the *memory cycle time*, describes how frequently you can repeat a memory reference. They sound identical, but they are often quite different due to phenomena such as the need to refresh DRAM cells.

There is quite a gap between the speed of memory and the speed of microprocessors. In the early 1980s, the access time of commonly available DRAM was about 200 ns, which was shorter than the clock cycle of the commonly used 4.77 MHz (210 ns) microprocessors of the day. Fastforwarding two decades, the clock cycle time of the average home microprocessor is down to about a nanosecond (1 GHz), but memory access times are hovering around 50 ns.

Many different kinds of memory modules have been developed to improve system performance. I'll present a few here.

- The oldest option currently in use is *fast-page mode* (FPM) memory. It implements the ability to read a full page (4 KB or 8 KB) of data during a single memory access cycle.
- A refinement of this technique goes into *extended data output* (EDO) memory. The refinements are largely based on electrical modifications.
- A more revolutionary change was implemented in *synchronous DRAM* (SDRAM) memory, which uses a clock to synchronize the input and output of signals. This clock is coordinated with the CPU clock, so the timings of all the components are synchronized. SDRAM also implements two memory banks on each module, which essentially doubles the memory throughput; it also allows multiple memory requests to be pending at once. A variation on SDRAM, called *double-data rate SDRAM* (DDR SDRAM) is able to read data on both the rising and falling edges of the clock, which doubles the data rate of the memory chip.

  SDRAM modules are usually described as PC66, PC100, or PC133, which refers to the clock rate they are driven at: 66 MHz (15 ns), 100 MHz (10 ns), or 133 MHz (8 ns), respectively. DDR SDRAM modules, however, are usually referred to by their peak throughput; a PC2100 module, for example, is theoretically capable of about 2.1 GB/sec peak throughput.
- *Direct Rambus* (RDRAM) memory, however, takes an entirely different approach. It uses a narrow (16 bits wide) but extremely fast (600-800 MHz) path to memory, and allows sophisticated pipelining of operations. It is widely used in high-performance embedded applications (e.g., Sony's PlayStation 2 and the Nintendo64 consoles). RDRAM modules are usually called PC600, PC700, or PC800, which refers to their clock rate.

Workstations and servers tend to be able to *interleave* memory across banks, which is conceptually similar to disk striping (see); if we have 128 MB of memory configured as four 32 MB modules, we store one bit on each module in turn, rather than the first 32 MB on the first module, the second 32 MB on the second module, etc. This allows much higher performance because of the avoidance of cycle time delays. Interleaving is almost always done in power-of-two increments.[3]

It can also lead to confusion. Let's say we have a system with seven memory banks filled. The memory controller may well decide to make a four-way interleave, a two-way interleave, and a one-way interleave, which would mean that a memory-intensive process's performance would depend on where exactly it fell in memory. The best resource on how to configure your system's memory for optimum interleaving is the hardware vendor.

Virtual Memory Architecture

A *virtual memory system* exists to provide a framework for the system to manage memory on the behalf of various processes. The virtual memory system provides two primary benefits. It allows software developers to write to a simple memory model, which shields the programmer from the memory subsystem's hardware architecture and allows the use of memory sizes substantially greater than

physical memory through backing stores. It also permits processes to have nonfragmented address spaces, regardless of how physical memory is organized or fragmented. In order to implement such a scheme, four key functions are required.

First, each process is presented with its own *virtual address space*; that is, it can "see" and potentially use a range of memory. This range of memory is equal to the maximum address size of the machine. For example, a process running on a 32-bit system will have a virtual address space of about 4 GB (232). The virtual memory system is responsible for managing the associations between the used portions of this virtual address space into physical memory.

Second, several processes might have substantial sharing between their address spaces. For example, let's say that two copies of the shell `/bin/csh` are running. Both copies will have separate virtual address spaces, each with their own copy of the executable itself, the *libc* library, and other shared resources. The virtual memory system transparently maps these shared segments to the same area of physical memory, so that multiple copies are not stored. This is analogous to making a hard link in a filesystem instead of duplicating the file.

However, sometimes physical memory will become insufficient to hold the used portions of all virtual address spaces. In this case, the virtual memory system selects less frequently used portions of memory and pushes them out to secondary storage (e.g., disk) in order to optimize the use of physical memory.

Finally, the virtual memory system plays one of the roles of an elementary school teacher: keeping the children in his care from interfering with each other's private things. Hardware facilities in the memory management unit perform this function by preventing a process from accessing memory outside its own address space.

Pages

Just like energy in quantum mechanics, memory is quantized; that is, it is organized into indivisible units. These units are called *pages*. The exact size of a page varies from system to system and is dependent on the processor's *memory management unit* (MMU) implementation. Large page sizes reduce MMU activity by reducing the number of page faults, and save kernel memory. However, they also waste some memory, since the system can only deal with memory in page-sized segments. A request for less than one full page of memory will be answered with the allocation of a full page. Typically, page sizes are 4 KB or 8 KB. On Solaris systems, you can find the page size via `/usr/bin/pagesize` or *getpagesize(3C)*, while on Linux it is defined in the kernel header file *asm/param.h* as `EXEC_PAGESIZE`.

Page Sizes on Various Architectures

There are only three commonly encountered microprocessor designs that implement an 8 KB page size: the DEC Alpha, the original Sun SPARC processors (e.g., the Ross RT601/Cypress CY7C601/Texas Instruments TMS390C601A, which were used in the SPARCstation 2), and the Sun UltraSPARC designs. The Intel 80x86, the MIPS processors used in SGI systems, the Motorola/IBM PowerPC, and Sun's microSPARC and SuperSPARC series processors all use a 4 KB page size.

Segments

The pages that compose a process are grouped into several segments. Each process has at least four of these segments:

Executable text
: Consists of the actual executable instructions in the binary. This is mapped from the on-disk binary and is read-only.

E*xecutable data*
: Contains initialized variables present in the executable. This is mapped from the on-disk binary, but has permissions set to read/write/private (the private mapping ensures that changes to these variables during runtime are not reflected in the on-disk file, or in other processes sharing the same executable).

H*eap space*
: Consists of memory allocated by means of `malloc(3)`. This is called anonymous memory, because it has no mapping in the filesystem (we'll discuss it further in "Anonymous memory" later in this chapter).

S*tack*
: Also allocated from anonymous memory.

Estimating Memory Requirements

Sometimes you have it easy, and a system only needs to run one commercial software package, for which the vendor has nicely told you precisely how much memory you need for optimal performance. Unfortunately, the real world is rarely so nice: the systems administrator in a typical computing environment has hundreds of different processes to worry about. Rather than try and solve every possible problem, we'll work out a general mechanism for thinking about how much memory is required for a given system.

The most important thing is to establish what a reasonable workload is on the system. Let's say that you have a system that's being used for serving typical low-end users: they run a shell, maybe an editor, a mailreader (probably pine or elm), or perhaps a newsreader. If you've been carefully monitoring your current environment, you'll be able to get a handle on how many of each sort of process is running at peak usage times. If you haven't, or if you're starting from scratch, you'll need to make a reasonable guess. Let's say that you've decided that there will be 50 users logged in at peak hours, and the usage pattern works out like this:

- 5 invocations each of `ksh` and `csh`, 40 of `tcsh`
- 25 `pine` processes and 10 `elm` processes
- Other applications that defy categorization

We'll work through calculating the memory requirements of the `csh` processes. If we run *pmap* on a representative process, we get output that resembles this:

```
% pmap -x 7522
7522:   -csh
Address   Kbytes Resident Shared Private Permissions       Mapped File
0803C000      48       48      -      48 read/write/exec    [ stack ]
08048000     116      116    116       - read/exec         csh
08065000      16       16      -      16 read/write/exec   csh
08069000      72       72      -      72 read/write/exec    [ heap ]
DFF19000     548      444    408      36 read/exec         libc.so.1
DFFA2000      28       28      4      24 read/write/exec   libc.so.1
DFFA9000       4        4      -       4 read/write/exec    [ anon ]
DFFAB000       4        4      4       - read/exec
libmapmalloc.so.1
DFFAC000       8        8      -       8 read/write/exec
libmapmalloc.so.1
DFFAF000     148      136    136       - read/exec         libcurses.so.1
DFFD4000      28       28      -      28 read/write/exec   libcurses.so.1
DFFDB000      12        -      -       - read/write/exec    [ anon ]
DFFDF000       4        4      -       4 read/write/exec    [ anon ]
DFFE1000       4        4      4       - read/exec         libdl.so.1
DFFE3000     100      100    100       - read/exec         ld.so.1
DFFFC000      12       12      -      12 read/write/exec   ld.so.1
--------  ------  ------  ------  ------
total Kb    1152     1024     772     252
```

It looks like each process is taking about 1,150 KB of memory, with 1,024 KB resident in memory. Of that 1,024 KB, 772 KB is shared with other processes, and 252 KB is private. So, for five invocations, we could roughly estimate memory usage at roughly 2 MB (about 770 KB shared, plus five times about 250 KB). These are back-of-the-envelope calculations, but they are remarkably good at giving you an idea how much memory you need.

Remember, though, that memory is consumed by three other things (the filesystem cache, intimately shared memory, and the kernel) aside from processes! Unless you're running Oracle or another database application, you probably won't need to worry about intimately shared memory. As a rule of thumb, you can safely figure on about 32 MB for the kernel and other applications, plus another 16 MB if you're running a windowing system. If your users only access a few hundred megabytes of data, but access it frequently, it may be advisable to buy sufficient memory to cache the entire data set. This will drastically improve I/O performance to that information.

Address Space Layout

The exact details of a process's address space vary depending on the architecture it is implemented on. For Solaris systems, there are essentially four address space layouts:

- SPARC V7 32-bit combined kernel and process address space (used on `sun4c`, `sun4m`, and `sun4d` architectures)
- SPARC V9 32-bit separated kernel and process address space (used on `sun4u` machines)
- SPARC V9 64-bit separated kernel and process address space (used on `sun4u` machines)

- Intel IA-32 32-bit combined kernel and process address space (used on Solaris for Intel systems)

The SPARC V7 32-bit combined model maps the kernel's address space into the top of the process's address space. This means that the virtual address space available to the process is restricted by the amount consumed by the kernel's address space (256 MB on `sun4c` and `sun4m` architectures, and 512 MB on `sun4d` architectures). The kernel's address space is protected from the user's process by means of the the processor's privilege levels. The stack begins just beneath the kernel at address `0xEFFFC000` (`0xDFFFE000` on `sun4d` systems) and grows downwards to address `0xEF7EA000` (`0xDF7F9000` on `sun4d`), where libraries are loaded into memory. The text and data portions of the executable are loaded into the bottom of the address space, and the heap grows from the top of those upwards towards the libraries.

The UltraSPARC microprocessor architecture allows the kernel its own private address space, which removes the size limit on the kernel's address space.[4] Thus, for the 32-bit `sun4u` memory model, the stack begins at address `0xFFBEC000` and grows downwards until `0xFF3DC000` (the small space at the absolute top of the address space is reserved for the OpenBoot PROM). Otherwise, it is the same as the SPARC V7 models.

Although the UltraSPARC processor supports the SPARC V9 64-bit mode, the UltraSPARC-I and UltraSPARC-II processor implementations allow only 44-bit physical memory addresses. This creates a "hole" in the middle of the virtual address space, which spans addresses `0xFFFFF7FF.FFFFFFFF` to `0x00000800.00000000`.

The Intel architecture address space is similar to the sun4c/sun4m SPARC V7 model, in that is does not separate user space from kernel space. However, it has a significant difference: the stack is mapped beneath the executable segments (starting at address `0x8048000`) and permitted to grow downwards to the very bottom of the address space.

The Free List

The free list is the mechanism by which the system dynamically manages the flow of memory between processes. Memory is taken from the free list by processes, and returned to it when the process exits or by the action of the page scanner, which tries to ensure that there is a small amount of memory free for immediate use at all times. Every time memory is requested, a *page fault* is incurred. There are three kinds of faults, which we'll discuss in more detail later in this section:

M*inor page fault*
: Occurs every time a process needs some memory, or when a process tries to access a page that has been stolen by the page scanner but not recycled.

M*ajor page fault*
: Occurs when a process tries to access a page that has been taken from it by the page scanner, completely recycled, and is now being used by another process. A major page fault is always preceded by a minor page fault.

C*opy-on-write fault*
: Caused by a process trying to write to a page of memory that it shares with other processes.

Let's examine how the free list is managed under a Solaris system.

When a system boots, all of its memory is formed into pages, and a kernel data structure is created to hold their states. The kernel reserves a few megabytes of memory for itself and releases the rest to the free list. At some point, a process will request memory, and a minor page fault occurs. A page is then taken from the free list, zeroed, and made available to the process. This sort of behavior, in which memory is given out on an "as-needed" basis, has traditionally been called *demand paging*.

> **TIP:** Pages are always taken from the head of the free list.

When the free list shrinks to a certain size (set to `lotsfree` in units of pages), the kernel wakes up the page scanner (also called the *pagedaemon*), which begins to search for pages that it can steal to replenish the free list. The page scanner implements a two-step algorithm order to avoid stealing pages that are being frequently accessed. The pagedaemon looks through memory in physical memory order and clears the MMU reference bit for each page. When a page is accessed, this bit is set. The page scanner then delays for a short while, waiting for pages to be accessed and their reference bits cleared. This delay is controlled by two parameters:

- `slowscan` is the initial scan rate. Increasing this value causes the page scanner to run less unnecessary jobs, but do more work.
- `fastscan` is the scan rate when the free list is completely empty.

The pagedaemon then goes through memory again. If a particular page still has the reference bit cleared, then the page hasn't been touched, so it is stolen for recycling. Think of memory as a circular track. A train rides on this track; as the engine in the front of the

train passes a tie, that tie's reference bit is cleared. When the caboose gets around to a tie, if that tie's reference bit is still cleared, that page is recycled.

Some pages are very resistant to recycling, such as those that belong to the kernel and those that are shared between more than eight processes (this helps to keep shared libraries in memory). If the stolen page does not contain cached filesystem data, then it is moved to the page-out queue, marked as pending I/O, and eventually written out to swap space with a cluster of other pages. If the page is being used for filesystem caching, it is not written to disk. In either case, the page is placed on the end of the free list, but not cleared; the kernel remembers that this particular page still stores valid data.

If the average size of the free list over a 30-second interval is less than `desfree`, the kernel begins to take desperate measures to free memory: inactive processes are swapped out, and pages are written out immediately rather than being clustered. If the 5-second average of free memory is less than `minfree`, active processes start to be swapped out. The page scanner stops looking for memory to reclaim when the size of the free list increases above `lotsfree`.

The page scanner is governed by a parameter, `maxpgio`, which limits the rate at which I/O occurs to the paging devices. It is, by default, set very low (40 pages per second on `sun4c`, `sun4m`, and `sun4u` architectures, and 60 pages per second on `sun4d`), in order to prevent saturating the paging device with I/Os. This rate is inadequate for a modern system with fast disks, and should be set to 100 times the number of spindles of configured swap space. If a process tries to access a recycled page, another minor page fault is incurred. There are now two possibilities:

- The page that was recycled has been placed back on the free list, but hasn't been reused yet. The kernel gives the page back to the process.
- The page has been recycled completely, and is now in use by another process. A major page fault is incurred and the paged-out data is read into a new page, taken from the free list.

Since pages can be shared between multiple processes, special precautions need to be taken in order to ensure that changes to a page are maintained locally to that process. If a process tries to write to a shared page, it incurs a *copy-on-write fault*.[5] A page is taken from the free list, and a copy of the original shared page is made for the process. When a process completes, all of its nonshared pages are returned to the free list.

Virtual memory management in Linux

This mechanism is used in most modern memory management systems, but often with some minor changes. Here, I discuss the Linux 2.2 kernel, as at the time of this writing the 2.4 kernel was still undergoing significant revisions to the VM subsystem. The Linux kernel implements a different set of kernel variables that tune memory performance a little bit differently.

When the size of a Linux system's free list drops below the value of `freepages.high`, the system begins to gently page. When the available memory drops below the `freepages.low` value, the system pages heavily, and if the free list shrinks beneath `freepages.min`, only the kernel can allocate more memory. The `freepages` tunables are located in the */proc/sys/vm/freepages* file, which is formatted as `freepages.min freepages.low freepages.high`.

The pagedaemon under Linux is called `kswapd`, which frees as many pages as necessary to get the system's free list back over the `freepages.high` mark. `kswapd`'s behavior is controlled by three parameters, called `tries_base`, `tries_min`, and `swap_cluster`, located in that order in the */proc/sys/vm/kswapd* file. The most important tunable is `swap_cluster`, which is the number of pages that `kswapd` will write out in a single turn. This value should be large enough so that `kswapd` does its I/O in large chunks, but small enough that it won't overload the disk I/O request queue. If you find yourself getting short on memory and paging heavily, you probably will want to experiment with increasing this value, to buy yourself more bandwidth to the paging space.

When a page fault is taken in Linux, multiple pages are actually read in to avoid multiple short trips to disk; the precise number of pages that are brought into memory is given by 2*n*, where n is the value of the `page-cluster` tunable (the only value in the */proc/sys/vm/page-cluster* tunable file). It is pointless to set this value above 5.

There are two important things to take from the page lifecycle described here. The first is the manner by which the page scanner operates, and the parameters that define its behavior. The page scanner manages the ebb and flow of the free list by controlling paging, which can have a tremendous impact on performance. The second is the way pages are allocated and released; minor page faults occur when a process requests a page of memory, and major page faults occur if that requested page doesn't contain the data the process expects it to. Table 4-1 summarizes the kernel variables related to the behavior of the virtual memory system.

Table 4-1: A summary of memory-related kernel variables

Operating		

system	Variable	Effect
Solaris	`lotsfree`	The target size of the free list, in pages. Set by default, one sixty-fourth of physical memory, with a minimum size of 512 KB (64 or 128 pages on an 8 KB or 4 KB page size system, respectively). The Linux equivalent is `freemem/high`.
	`desfree`	The minimum pages of free memory, taken over an interval of 30 seconds, before swapping starts. Set to one-half of `lotsfree` by default. Roughly equivalent to Linux `freemem/low`.
	`minfree`	The threshhold of minimum free memory, taken over a 5-second interval, before active swapping of processes starts. Represented in pages. Set by default to one-half of `desfree`. There is no equivalent parameter under Linux.
	`slowscan`	The slowest scan rate (e.g., when scanning first starts) in pages. Set by default to one sixty-fourth of the size of physical memory, with a minimum of 512 KB; that is, 64 pages on an 8 KB page size system, or 128 pages on a 4 KB page size system. There is no Linux equivalent.
	`fastscan`	The fastest scan rate (e.g., with the free list completely empty) in pages. Set by default to half the physical memory, with a maximum of 64 MB; that is, 8,192 pages on an 8 KB page size system, or 16,384 pages on a 4 KB page size system. There is no Linux equivalent.
	`maxpgio`	The maximum number of I/Os per second that will be queued by the page scanner. Set by default to 40 on all systems except those based on the `sun4d` architecture, which has a default setting of 60. There is no Linux equivalent.
Linux (2.2.x)	`freepages.high`	The target size of the free list in pages. When memory drops below this level, the system starts to gently page. Set by default to 786. The Solaris equivalent is `lotsfree`.
	`freepages.low`	The threshhold for aggressive memory reclamation in pages. By default, this is set to 512. A rough Solaris equivalent is `minfree`.
	`freepages.min`	The minimum number of pages on the free list. When this value is reached, only the kernel can allocate more memory. Set to 256 by default. There is no equivalent parameter under Solaris.
	`page-cluster`	The number of pages that are read in on a page fault is given by 2*page-cluster*. The default value is 4. There is no equivalent parameter under Solaris.

Page Coloring

The precise way that pages are organized in processor caches can have a dramatic effect on application performance. The optimal placement of pages depends on how the application utilizes memory--some applications access memory almost randomly, whereas others access memory in a strict sequential order--so there is no single algorithm that provides universally optimal results.

The free list is actually organized into a number of specifically colored bins. The number of bins is given by the physical second-level cache size divided by the page size (for example, a system with a 64 KB L2 cache and an 8 KB page size would have 8 bins). When a page is returned to the free list, it is assigned to a specific bin. When a page is required, one is taken from a specific colored bin. That choice is made based on the virtual address of the requested page. (Recall that pages exist in physical memory, and are physically addressable, but processes only understand virtual addresses; a process incurs a page fault on a virtual address in its own address space, and that page fault is filled by a page with a physical address.) The algorithms we are discussing here decide how pages are assigned colors. There are three available page coloring algorithms in Solaris 7 and later:

- The default algorithm (numbered 0) uses a hashing algorithm against the virtual address, in order to distribute pages as evenly as possible.
- The first optional algorithm (numbered 1) assigns colors to pages so that physical addresses map directly to virtual addresses.
- The second optional algorithm (numbered 2) assigns pages to bins in a round robin mechanism.

Another algorithm called Kessler's Best Bin (numbered 6) was supported on Ultra Enterprise 10000 systems running Solaris 2.5.1 and 2.6 only. It used a history-based mechanism to try and assign pages to the least-used bins.

The default paging algorithm was chosen because it consistently provides good performance. You are not likely to see any performance improvement from tuning these parameters on typical commercial workloads. However, scientific applications may see a significant performance gain--some testing will reveal the page coloring algorithm that fits best with your application's memory usage patterns. You can change the algorithm by setting the system parameter `consistent_coloring` to the number of the algorithm you'd like to use.

Transaction Lookaside Buffers (TLB)

While we're discussing the physical to virtual address relationship, it's important to mention the *transaction lookaside buffer* (TLB). One of the jobs of the hardware memory management unit is to translate a virtual address, as provided by the operating system, into a physical address. It does this by looking up entries in a page translation table. The most recent translations are cached in the translation lookaside buffer. Intel and older SPARC processors (pre-UltraSPARC) use hardware mechanisms to populate the TLB, whereas the UltraSPARC architecture uses software algorithms. While a detailed discussion of the impact of the TLB on performance is a bit beyond the scope of this book, you should be generally aware of what a TLB is and what it does.

Paging and Swapping

Paging and swapping are terms that are often used interchangeably, but they are quite distinct. A system that is *paging* is writing selected, infrequently used pages of memory to disk, while a system that is *swapping* is writing entire processes from memory to disk. Let's say that you are working on your automobile, and you only have a small amount of space available for tools. Paging is equivalent to putting the 8 mm socket back in the toolchest so you have enough room for a pair of pliers; swapping is like putting your entire socket set away.

Many people feel that their systems should never nontrivially page (that is, perform paging on a pre-Solaris 8 system that is not simply filesystem activity). It's important to realize that paging and swapping allow the system to continue getting work done despite adverse memory conditions. Paging is not necessarily indicative of a problem; it is the action of the page scanner to try and increase the size of the free list by moving inactive pages to disk. A process, as a general rule, spends about 80% of its time running about 20% of its code; since the entire process doesn't need to be in memory at once, writing some pages out to disk won't affect performance substantially. Performance only begins to suffer when a memory shortage continues or worsens.

The Decline and Fall of Interactive Performance

Historically, Unix systems implemented a time-based swapping mechanism, whereby a process that was idle for more than 20 seconds would be swapped out; this isn't done anymore. Swapping is now used only to address the most severe memory shortages. If you come across a system where `vmstat` reports a nonzero swap queue, the only conclusion you can draw is that, at some indeterminate time in the past, the system was short enough on memory to swap out a process.

Memory shortages also tend to appear worse than they are because of the nature of the memory reclamation mechanism. When a system is desperately swapping jobs out, it tries to avoid a major performance decrease by keeping active jobs in memory for as long as possible. Unfortunately, programs that directly interact with users (such as shells, editors, or anything else that is dependent on user-supplied input) are inactive while waiting for the user to type something. As a result, these interactive processes are likely to be targeted as good places for memory reclamation efforts to be targeted. For example, when you pause momentarily in tying a command (such as when you're looking through the process table trying to find the ID of the process that is hogging all your memory so you can

kill it), your shell will have to make the long trip from disk back into memory before your characters can be echoed back. Even worse, the disk subsystem is probably under heavy load from all the paging and swapping activity! The upshot is that in memory shortages, interactive performance falls through the floor.

This situation degrades even further if memory consumption continues. The system starts to slow down as disks become overloaded with paging and swapping, and the load average spirals higher; memory-based constraints quickly turn into I/O-based constraints.

Swap Space

Swap space (or, more accurately, *paging space*, since almost all of this sort of activity is paging rather than swapping) is a topic that tends to confuse people. Swap space serves three functions:

- As a place to write pages of private memory (a *paging store*)
- As a place for anonymous memory storage
- As a place to store crash dumps[6]

This space is allocated from both spare physical memory and from a swap space on disk, be it a dedicated partition or a swapfile.

Anonymous memory

Swap space for anonymous memory is used in two stages. Anonymous memory reservations are taken out of disk-based swap, but allocations are taken out of physical memory-based swap. When anonymous memory is requested (via the *malloc(3C)* system call, for example), a reservation is made against swap, and a mapping is made against */dev/zero*. Disk-based swap is used until none remains, in which case physical memory is used instead. Memory space that is mapped but never used stays reserved. This is common behavior in large database systems, and explains why applications such as Oracle require large amounts of disk-based swap, even though they are unlikely to allocate the entire reserved space.

When reserved pages are accessed for the first time, physical pages are taken from the free list, zeroed, and allocated rather than reserved. Should a page of anonymous memory be stolen by the page scanner, the data is written to disk-based swap space (that is, the allocation is moved from memory to disk), and the memory is freed for reuse.

Sizing swap space

There are many rules of thumb for how much swap space should be configured on a system, ranging from four times the amount of physical memory to half the amount of physical memory, and none of them are particularly good.

Under Solaris, one tool to get a long term picture of swap usage is `/usr/sbin/swap -s`. Here's an example from a workstation running Solaris 7 with 128 MB of physical memory and a 384 MB swap partition:

```
% swap -s
total: 12000k bytes allocated + 3512k reserved = 15512k used, 468904k available
```

Unfortunately, the names of these fields are misleading, if not downright incorrect. A more accurately labeled version would read:

```
total: 12000k bytes allocated + 3512k unallocated = 15512k reserved, 468904k available
```

When the available swap space reaches zero, the system will no longer be able to use more memory (until some is freed). From this report, it is obvious that the system has plenty of swap space configured.

You can also use `/usr/sbin/swap -l` to find out how much swap space is in use. Here's an example from the same system:

```
% swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c1t1d0s1   32,129     16 788384 775136
```

In this case, there is a single swap device (a dedicated partition) with a total area of 688,384 512-byte blocks (about 384 MB). At the time this measurement was taken, virtually all of the swap space was free.

One tool you can use to monitor swap space usage over time is `/usr/bin/sar -r` *interval*, with *interval* specified in seconds:

```
% sar -r 3600
SunOS fermat 5.7 Generic sun4u    06/01/99
```

```
00:00:00 freemem freeswap
00:00:00     679   935313
01:00:00     680   937184
02:00:00     680   937184
```

`sar` reports the free memory in pages, and the free swap in disk blocks (really the amount of available swap space).

On Linux systems, you can get similar information from the file */proc/meminfo*:

```
% cat /proc/meminfo | head -3 | grep -vi mem
        total:    used:    free:  shared: buffers:  cached:
Swap: 74657792 18825216 55832576
```

In general, if more than about half of your swap area is in use, you should consider increasing it. Disk is very cheap; skimping on swap space is only going to hurt you.

Organizing swap space

In order to minimize the impact of paging as much as possible, swap areas should be configured on the fastest possible disks. The swap partition should be on a low numbered slice (traditionally, slice one, with the root filesystem on slice zero); we'll explain why in Chapter 5. There is absolutely no reason, from a performance point of view, to place swap areas on slow disks, fast disks being accessed through slow controllers, fast disks that are already heavily loaded with I/O activity, or to have more than one swap area on a disk. The best strategy is to have a dedicated swapping area with multiple fast disks and controllers. If it is possible, try to put the swap area on a fast, lightly used disk. If it is not possible, however, try and put the swap file on the most heavily used partition. This minimizes the seek time to the disk; for a detailed discussion, see .

Solaris allows you to use remote files as swapping areas via NFS. This isn't a good way to achieve reasonable performance on modern systems. If you're configuring a diskless workstation, you have no choice unless you can survive without disk-based swap space. Measuring the level of I/O activity on a per-partition and per-disk basis is a topic that we'll cover in more detail in . The single most important issue in swap area placement is to put swap areas on low numbered partitions on fast, lightly used disks.

Swapfiles

Sometimes, you need to create a swap area in an "emergency" setting. The most effective way of doing this on a Solaris system is to use a *swapfile*, which is a file on disk that the system treats as swap space. In order to create one, you must first create an empty file with `/usr/sbin/mkfile`, then use `/usr/bin/swap -a` on the swapfile. Here's an example:

```
# mkfile 64m /swapfile
# /usr/bin/swap -a /swapfile
# /usr/bin/swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c0t0d0s0   32,0      16 262944 232816
/swapfile              -      16  65520  65520
```

You may alternately specify the size of the swapfile to `mkfile` in blocks or kilobytes by specifying the units as `b` or `k`, respectively. You can remove the file from the swap space via `/usr/bin/swap -d` *swapfile*. Note that the swapfile will not be automatically activated at system boot.

Consumers of Memory

Memory is consumed by four things: the kernel, filesystem caches, processes, and intimately shared memory. When the system starts, it takes a small amount (generally less than 4 MB) of memory for itself. As it dynamically loads modules and requires additional memory, it claims pages from the free list. These pages are locked in physical memory, and cannot be paged out except in the most severe of memory shortages. Sometimes, on a system that is very short of memory, you can hear a pop from the speaker. This is actually the speaker being turned off as the audio device driver is being unloaded from the kernel. However, a module won't be unloaded if a process is actually using the device; otherwise, the disk driver could be paged out, causing difficulties. Occasionally, however, a system will experience a *kernel memory allocation error*. While there is a limit on the size of kernel memory,[7] the problem is caused by the kernel trying to get memory when the free list is completely exhausted. Since the kernel cannot always wait for memory to become available, this can cause operations to fail rather than be delayed. One of the subsystems that cannot wait for memory is the streams facility; if a large number of users try to log into a system at the same time, some logins may fail. Starting with Solaris 2.5.1, changes were made to expand the free list on large systems, which helps prevent the free list from ever being totally empty.

Processes have private memory to hold their stack space, heap, and data areas. The only way to see how much memory a process is actively using is to use `/usr/proc/bin/pmap -x process-id`, which is available in Solaris 2.6 and later releases.

Intimately shared memory is a technique for allowing the sharing of low-level kernel information about pages, rather than by sharing the memory pages themselves. This is a significant optimization in that it removes a great deal of redundant mapping information. It is of primary use in database applications such as Oracle, which benefit from having a very large shared memory cache. There are three special things worth noting about intimately shared memory. First, all the intimately shared memory is locked, and cannot ever be paged out. Second, the memory management structures that are usually created independently for each process are only created once, and shared between all processes. Third, the kernel tries to find large pieces of contiguous physical memory (4 MB) that can be used as large pages, which substantially reduces MMU overhead.

Filesystem Caching

The single largest consumer of memory is usually the filesystem-caching mechanism. In order for a process to read from or write to a file, the file needs to be buffered in memory. When this is happening, these pages are locked in memory. After the operation completes, the pages are unlocked and placed at the bottom of the free list. The kernel remembers the pages that store valid cached data. If the data is needed again, it is readily available in memory, which saves the system an expensive trip to disk. When a file is deleted or truncated, or if the kernel decides to stop caching a particular inode, any pages caching that data are placed at the head of the free list for immediate reuse. Most files, however, only become uncached upon the action of the page scanner. Data that has been modified in the memory caches is periodically written to data by `fsflush` on Solaris and `bdflush` on Linux, which we'll discuss a little later.

The amount of space used for this behavior is *not* tunable in Solaris; if you want to cache a large amount of filesystem data in memory, you simply need to buy a system with a lot of physical memory. Furthermore, since Solaris handles all its filesystem I/O by means of the paging mechanism, a large number of observed page-ins and page-outs is completely normal. In the Linux 2.2 kernel, this caching behavior is tunable: only a specific amount of memory is available for filesystem buffering. The `min_percent` variable controls the minimum percentage of system memory available for caching. The upper bound is not tunable. This variable can be found in the */proc/sys/vm/buffermem* file. The format of that file is `min_percent max_percent borrow_percent`; note that `max_percent` and `borrow_percent` are not used.

Filesystem Cache Writes: fsflush and bdflush

Of course, the caching of files in memory is a huge performance boost; it often allows us to access main memory (a few hundred nanoseconds) when we would otherwise have to go all the way to disk (tens of milliseconds). Since the contents of a file can be operated upon in memory via the filesystem cache, it is important for data-reliability purposes to regularly write changed data to disk. Older Unix operating systems, like SunOS 4, would write the modified contents of memory to disk every 30 seconds. Solaris and Linux both implement a mechanism to spread this workload out, which is implemented by the `fsflush` and `bdflush` processes, respectively.

This mechanism can have substantial impacts on a system's performance. It also explains some unusual disk statistics.

Solaris: fsflush

The maximum age of any memory-resident modified page is set by the `autoup` variable, which is thirty seconds by default. It can be increased safely to several hundred seconds if necessary. Every `tune_t_fsflushr` seconds (by default, every five seconds), `fsflush` wakes up and checks a fraction of the total memory equal to `tune_t_fsflushr` divided by `autoup` (that is, by default, five-thirtieths, or one-sixth, of the system's total physical memory). It then flushes any modified entries it finds from the inode cache to disk; it can be disabled by setting `doiflush` to zero. The page-flushing mechanism can be totally disabled by setting `dopageflush` to zero, but this can have serious repercussions on data reliability in the event of a crash. Note that `dopageflush` and `doiflush` are complimentary, not mutually exclusive.

Linux: bdflush

Linux implements a slightly different mechanism, which is tuned via the values in the */proc/sys/vm/bdflush* file. Unfortunately, the tunable behavior of the *bdflush* daemon has changed significantly from the 2.2 kernels to the 2.4 kernels. I discuss each in turn.

In Linux 2.2, if the percentage of the filesystem buffer cache that is "dirty" (that is, changed and needs to be flushed) exceeds `bdflush.nfract`, then `bdflush` wakes up. Setting this variable to a high value means that cache flushing can be delayed for quite a while, but it also means that when it does occur, a lot of disk I/O will happen at once. A lower value spreads out disk activity more evenly. `bdflush` will write out a number of buffer entries equal to `bdflush.ndirty`; a high value here causes sporadic, bursting I/O,

but a small value can lead to a memory shortage, since `bdflush` isn't being woken up frequently enough. The system will wait for `bdflush.age_buffer` or `bdflush.age_super`, in hundredths of a second, before writing a dirty data block or dirty filesystem metadata block to disk. Here's a simple Perl script for displaying, in a pretty format, the values of the `bdflush` configuration file:

```
#!/usr/bin/perl
my ($nfract, $ndirty, $nrefill, $nref_dirt, $unused, $age_buffer, $age_super, $unused, $unused) = split (/\
s+/, `cat /proc/sys/vm/bdflush`, 9);
print "Current settings of bdflush kernel variables:\n";
print "nfract\t\t$nfract\tndirty\t\t$ndirty\tnrefill\t\t$nrefill\n\r";
print "nref_dirt\t$nref_dirt\tage_buffer\t$age_buffer\tage_super\t$age_super\n\r";
```

In Linux 2.4, about the only thing that didn't change was the fact that `bdflush` still wakes up if the percentage of the filesystem buffer cache that is dirty exceeds `bdflush.nfract`. The default value of `bdflush.nfract` (the first in the file) is 30%; the range is from 0 to 100%. The minimum interval between wakeups and flushes is determined by the `bdflush.interval` parameter (the fifth in the file), which is expressed in clock ticks.[8] The default value is 5 seconds; the minimum is 0 and the maximum is 600. The `bdflush.age_buffer` tunable (the sixth in the file) governs the maximum amount of time, in clock ticks, that the kernel will wait before flushing a dirty buffer to disk. The default value is 30 seconds, the minimum is 1 second, and the maximum is 6,000 seconds. The final parameter, `bdflush.nfract_sync` (the seventh in the file), governs the percentage of the buffer cache that must be dirty before `bdflush` will activate synchronously; in other words, it is the hard limit after which `bdflush` will force buffers to disk. The default is 60%. Here's a script to extract values for these `bdflush` parameters in Linux 2.4:

```
#!/usr/bin/perl
my ($nfract, $unused, $unused, $unused, $interval, $age_buffer, $nfract_sync, $u
nused, $unused) = split (/\s+/, `cat /proc/sys/vm/bdflush`, 9);
print "Current settings of bdflush kernel variables:\n";
print "nfract $nfract\tinterval $interval\tage_buffer $age_buffer\tnfract_sync $
nfract_sync\n";
```

If the system has a very large amount of physical memory, `fsflush` and `bdflush` (we'll refer to them generically as *flushing daemons*) will have a lot of work to do every time they are woken up. However, most files that would have been written out by the flushing daemon have already closed by the time they're marked for flushing. Furthermore, writes over NFS are always performed synchronously, so the flushing daemon isn't required. In cases where the system is performing lots of I/O but not using direct I/O or synchronous writes, the performance of the flushing daemons becomes important. A general rule for Solaris systems is that if `fsflush` has consumed more than five percent of the system's cumulative nonidle processor time, `autoup` should be increased.

Interactions Between the Filesystem Cache and Memory

Because Solaris has an untunable filesystem caching mechanism, it can encounter problems under some specific instances. The source of the problem is that the kernel allows the filesystem cache to grow to the point where it begins to steal memory pages from user applications. This behavior not only shortchanges other potential consumers of memory, but it means that the filesystem performance becomes dominated by the rate at which the virtual memory subsystem can free memory.

There are two solutions to this problem: priority paging and the cyclic cache.

Priority paging

In order to address this issue, Sun introduced a new paging algorithm in Solaris 7, called *priority paging*, which places a boundary around the filesystem cache.[9] A new kernel variable, `cachefree`, is created, which scales with `minfree`, `desfree`, and `lotsfree`. The system attempts to keep `cachefree` pages of memory available, but frees filesystem cache pages only when the size of the free list is between `cachefree` and `lotsfree`.

The effect is generally excellent. Desktop systems and on-line transaction processing (OLTP) environments tend to feel much more responsive, and much of the swap device activity is eliminated; computational codes that do a great deal of filesystem writing may see as much as a 300% performance increase. By default, priority paging has been disabled until sufficient end-user feedback on its performance is gathered. It will likely become the new algorithm in future Solaris releases. In order to use this new mechanism, you need Solaris 7, or 2.6 with kernel patch 105181-09. To enable the algorithm, set the `priority_paging` variable to 1. You can also implement the change on a live 32-bit system by setting the `cachefree` tunable to twice the value of `lotsfree`.

Cyclic caching

A more technically elegant solution to this problem has been implemented in Solaris 8, primarily due to efforts by Richard McDougall, a senior engineer at Sun Microsystems. No special procedures need be followed to enable it. At the heart of this mechanism is a straightforward rule: nondirty pages that are not mapped anywhere should be on the free list. This rule means that the free list now

contains all the filesystem cache pages, which has far-reaching consequences:

- Application startup (or other heavy memory consumption in a short period of time) can occur much faster, because the page scanner is not required to wake up and free memory.
- Filesystem I/O has very little impact on other applications on the system.
- Paging activity is reduced to zero and the page scanner is idle when sufficient memory is enabled.

As a result, analyzing a Solaris 8 system for a memory shortage is simple: if the page scanner reclaims any pages *at all*, there is a memory shortage. The mere activity of the page scanner means that memory is tight.

Interactions Between the Filesystem Cache and Disk

When data is being pushed out from memory to disk via `fsflush`, Solaris will try to gather modified pages that are adjacent to each other on disk, so that they can be written out in one continuous piece. This is governed by the `maxphys` kernel parameter. Set this parameter to a reasonably large value (1,048,576 is a good choice; it is the largest value that makes sense for a modern UFS filesystem). As we'll discuss in the section "RAID Recipes" in Chapter 6, a `maxphys` value of 1,048,576 with a 64 KB interlace size is sufficient to drive a 16-disk RAID 0 array to nearly full speed with a single file.

There is another case where memory and disk interact to give suboptimal performance. If your applications are constantly writing and rewriting files that are cached in memory, the in-memory filesystem cache is very effective. Unfortunately, the filesystem flushing process is regularly attempting to purge data out to disk, which may not be a good thing. For example, if your working set is 40 GB and fits entirely in available memory, with the default `autoup` value of 30, `fsflush` is attempting to synchronize up to 40 GB of data to disk every 30 seconds. Most disk subsystems cannot sustain 1.3 GB/second, which will mean that the application is throttled and waiting for disk I/O to complete, despite the fact that all of the working set is in memory!

There are three telltale signs for this case:

- `vmstat -p` shows very low filesystem activity.
- `iostat -xtc` shows constant disk write activity.
- The application has a high wait time for file operations.

Increasing `autoup` (to, say, 840) and `tune_t_fsflushr` (to 120) will decrease the amount of data sent to disk, improving the chances of issuing a single larger I/O (rather than many smaller I/Os). You will also improve your chances of seeing write cancellation, when not every modification to a file is written to disk. The flip-side is that you run a higher risk of losing data in the case of server failure.

Tools for Memory Performance Analysis

Tools for memory performance analysis can be classed under three basic issues: how fast is memory, how constrained is memory in a given system, and how much memory does a specific process consume? In this section, I examine some tools for approaching each of these questions.

Memory Benchmarking

In general, monitoring memory performance is a function of monitoring memory restraints. Tools for providing benchmarks as to how fast the memory subsystem is do exist; they are largely of academic interest, as it is unlikely that much tuning will be able to increase these numbers. The one exception to this rule is that users can carefully tune interleaving as appropriate. Most systems handle interleaving by purely physical means, so you may have to purchase additional memory: consult your system hardware manual for more information.[10] Nonetheless, it is often important to be aware of relative memory subsystem performance in order to make meaningful comparisons.

STREAM

The `STREAM` tool is simple; it measures the time required to copy regions of memory. This measures "real-world" sustainable bandwidth, not the theoretical "peak bandwidth" that most computer vendors provide. It was developed by John McCalpin while he was a professor at the University of Delaware.

The benchmark itself is easy to run in single-processor mode (the multiprocessor mode is quite a bit more complex; consult the benchmark documentation for current details). Here's an example from an Ultra 2 Model 2200:

```
$ ./stream
-------------------------------------------------------------
This system uses 8 bytes per DOUBLE PRECISION word.
-------------------------------------------------------------
Array size = 1000000, Offset = 0
Total memory required = 22.9 MB.
Each test is run 10 times, but only
the *best* time for each is used.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 40803 microseconds.
   (= 40803 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function      Rate (MB/s)   RMS time     Min time     Max time
Copy:         226.1804       0.0709       0.0707       0.0716
Scale:        227.6123       0.0704       0.0703       0.0705
Add:          276.5741       0.0869       0.0868       0.0871
Triad:        239.6189       0.1003       0.1002       0.1007
```

The benchmarks obtained correspond to the summary in Table 4-2.

Table 4-2: STREAM benchmark types

Benchmark	Operation	Bytes per iteration
Copy	`a[i] = b[i]`	16
Scale	`a[i] = q * b[i]`	16
Add	`a[i] = b[i] + c[i]`	24
Triad	`a[i] = b[i] + q * c[i]`	24

It is also interesting to note that there are at least three ways, all in common use, of counting how much data is transferred in a single operation:

Hardware method
: Counts how many bytes are physically transferred, as the hardware may move a different number of bytes than the user specified. This is because of cache behavior; when a store operation misses in a cache, many systems perform a *write allocate*, which means that the cache line containing that memory location is stored in the processor cache.[11]

Bcopy method
: Counts how many bytes get moved from one location in memory to another. If it takes your machine one second to read a certain number of bytes at one location and one more second to write the same number of bytes to another location, the resulting bandwidth is that number of bytes per second.

STREAM method
: Counts how many bytes the user asked to be read and how many bytes the user asked to be written. For the simple "copy" test, this is precisely twice the number obtained by the *bcopy* method. This is done because some of the tests perform arithmetic; it makes sense to count both the data read into and the data written back from the CPU.

One of the nice things about `STREAM` is that it uses the same method of counting bytes, all the time, so it's safe to make comparisons.

`STREAM` is available in source code, so that it can be easily compiled. A list of benchmarks are also available. The `STREAM` home page is located at http://www.streambench.org.

lmbench

Another tool for measuring memory performance is `lmbench`. While `lmbench` is capable of many sorts of measurements, we'll focus

on four specific ones. The first three measure bandwidth: the speed of memory reads, the speed of memory writes, and the speed of memory copies (via the *bcopy* method described previously). The last one is a measure of memory read latency. Let's briefly discuss what each of these benchmarks measures:

Memory copy benchmark
: Allocates a large chunk of memory, fills it with zeroes, and then measures the time required to copy the first half of the chunk of memory into the second half. The results are reported in megabytes moved per second.

Memory read benchmark
: Allocates a chunk of memory, fills it with zeroes, and then measures the time required to read that memory as a series of integer loads and adds; each 4-byte integer is loaded and added to an accumulator variable.

Memory write benchmark
: Allocates a chunk of memory, fills it with zeroes, and then measures the time needed to write that memory as a series of 4-byte integer stores and increments.

Memory latency benchmark
: Measures the time required to read a byte from memory. The results are reported in nanoseconds per load. The entire data memory hierarchy[12] is evaluated, including the latency to the primary caches, secondary caches, main memory, and the latency effects of a TLB miss. In order to derive the most information from the *lmbench* memory latency benchmarks, try plotting the data as a function of the latency versus the array size used for the test.

When you compile and run the benchmark, it will ask you a series of questions regarding what tests you would like to run. The suite is fairly well documented. The `lmbench` home page, which contains source code and more details about how the benchmark works, is located at http://www.bitmover.com/lm/lmbench/.

Examining Memory Usage System-Wide

It is important to understand a system's performance. The only way to get that understanding is through regular monitoring of data.

vmstat

`vmstat` is one of the most ubiquitous performance tools. There is one cardinal rule about `vmstat`, which you must learn and never forget: `vmstat` tries to present an average since boot on the first line. The first line of `vmstat` output is utter garbage and must be discarded.

With that in mind, Example 4-1 shows output under Solaris.

Example 4-1: vmstate output in Solaris

```
# vmstat 5
 procs     memory            page            disk          faults       cpu
 r b w   swap  free    re  mf pi po fr de sr s0 s1 s2 --  in   sy   cs  us sy id
 0 0 0  43248 49592   0   1  5  0  0  0  0  0  1  0  0  116  106   30  1  1  99
 0 0 0 275144 56936   0   1  0  0  0  0  0  2  0  0  0  120    5   19  0  1  99
 0 0 0 275144 56936   0   0  0  0  0  0  0  0  0  0  0  104    8   19  0  0 100
 0 0 0 275144 56936   0   0  0  0  0  0  0  0  0  0  0  103    9   20  0  0 100
```

The `r`, `b`, and `w` columns represent the number of processes that have been in the run queue, blocked for I/O resources (including paging), and processes that are runnable but swapped, respectively. If you ever see a system with a nonzero number in the `w` field, all that you can infer is that the system was, at some point in the past, low enough on memory to force swapping to occur. Here's the most important data you can glean from `vmstat` output:

`swap`
: The amount of available swap space, in KB.

`free`
: The amount of free memory (that is, the size of the free list) in KB. In Solaris 8, this number includes the amount of memory used for the filesystem cache; in prior releases, it does not, and therefore will be very small.

`re`
: The number of pages reclaimed from the free list. A page had been stolen from a process, but then reclaimed by the process before it was recycled and given to another process.

`mf`

The number of minor page faults. These are fast as long as a page is available from the free list, because they can be resolved without performing a page-in.

`pi`, `po`

Page-ins and page-outs, respectively, in KB per second.

`de`

The short term memory deficit. If the value is nonzero, memory has been vanishing fast lately, and extra free memory will be reclaimed with the expectation that it'll be needed soon.

`sr`

The scan rate activity of the pagedaemon, in pages per second. This is the most critical memory-shortage indicator. In Solaris systems prior to 8, if this stays about 200 pages per second for a reasonably long period of time, you need more memory. If you are running Solaris 8 and this number is nonzero, you are short of memory.

On Linux, the output is a little bit different:

```
% vmstat 5
 procs                  memory    swap        io     system         cpu
 r b w   swpd  free  buff cache  si  so   bi   bo   in   cs  us  sy  id
 1 0 0 18372  8088 21828 56704   0   0    1    7   13    6   2   1   6
 0 0 0 18368  7900 21828 56708   1   0    0    8  119   42   6   3  91
 1 0 0 18368  7880 21828 56708   0   0    0   14  122   44   6   3  91
 0 0 0 18368  7880 21828 56708   0   0    0    5  113   24   2   2  96
 0 0 0 18368  7876 21828 56708   0   0    0    4  110   27   2   2  97
```

Note that while the `w` field is calculated, Linux never desperation-swaps. `swpd`, `free`, `buff`, and `cache` represented the amount of virtual memory used, the amount of idle memory, the amount of memory used as buffers, and the amount used as cache memory, respectively. There is generally very little to be gathered from a Linux `vmstat` output. The most important thing to watch is probably the `si` and `so` columns, which indicate the amount of swap-ins and swap-outs. If these grow large, you probably need to increase the `kswapd swap_cluster` kernel variable to buy yourself some additional bandwidth to and from swap, or purchase more physical memory (see "Virtual memory management in Linux" earlier in this chapter).

sar

`sar` (for *system activity reporter*) is, like `vmstat`, an almost ubiquitous performance-monitoring tool. It is particularly useful in that it can be adapted to gather data on its own for later perusal, as well as doing more focused, short-term data collection. In general, its output is comparable with `vmstat`'s, although differently labeled.

In general, the syntax for invoking `sar` is `sar -flags` *interval number*. This causes a specific *number* of data points to be gathered every *interval* seconds. When looking at memory statistics, the most important flags are `-g`, `-p`, and `-r`. Here's an example of the output generated:

```
$ sar -gpr 5 100
SunOS islington.london-below.net 5.8 Generic_108528-03 sun4u    02/19/01

11:28:10  pgout/s ppgout/s pgfree/s pgscan/s %ufs_ipf
          atch/s  pgin/s ppgin/s  pflt/s  vflt/s slock/s
        freemem freeswap
...
11:28:50     0.00     0.00     0.00     0.00     0.00
           251.60    5.00    5.20 1148.20 2634.60    0.00
            86319  3304835
```

The most important output fields are summarized in Table 4-3.

Table 4-3: sar memory statistics fields

Flag	Field	Meaning
-g	`pgout/s`	Page-out requests per second
	`ppgout/s`	Pages paged out per second
	`pgfree/s`	Pages placed on the free list per second by the page scanner
	`pgscan/s`	Pages scanned per second by the page scanner
		The percentage of cached filesystem pages taken off the free list while they still contained valid data; these pages

	%ufs_ipf	are flushed and cannot be reclaimed (see)
-p	atch/s	Page faults per second that are satisfied by reclaiming a page from the free list (this is sometimes called an *attach*)
	pgin/s	The number of page-in requests per second
	ppgin/s	The number of pages paged in per second
	pflt/s	The number of page faults caused by protection errors (illegal access to page, copy-on-write faults) per second
-r	freemem	The average amount of free memory
	freeswap	The number of disk blocks available in paging space

memstat

Starting in the Solaris 7 kernel, Sun implemented new memory statistics to help evaluate system behavior. To access these statistics, you need a tool called `memstat`. While currently unsupported, it will hopefully be rolled into a Solaris release soon. This tool provides some wonderful functionality, and is available at the time of this writing from http://www.sun.com/sun-on-net/performance.html. Example 4-2 shows the sort of information it gives you.

Example 4-2: memstat

```
# memstat 5
memory ---------- paging----------executable- -anonymous---filesys  --  ---  cpu --
  free  re  mf  pi  po  fr de sr epi epo epf  api apo apf  fpi fpo fpf us sy wt  id
 49584   0   1   5   0   0  0  0   0   0   0    0   0   0    5   0   0  1  1  1  98
 56944   0   1   0   0   0  0  0   0   0   0    0   0   0    0   0   0  0  0  0 100
```

Much like `vmstat`, the first line of output is worthless and should be discarded. `memstat` breaks page-in, page-out, and page-free activity into three different categories: executable, anonymous, and file operations. Systems with plenty of memory should see very low activity in the `epf` and `apo` fields. Consistent activity in these fields indicates a memory shortage.

If you are running Solaris 7 or earlier and do not have priority paging enabled, however, executables and anonymous memory will be paged with even the smallest amount of filesystem I/O, once memory falls to or beneath the `lotsfree` level.

Examining Memory Usage of Processes

It's nice to know how much memory a specific process is consuming. Each process's address space, which is made up of many segments, can be measured in the following ways:

- The total size of the process address space (represented by SZ or SIZE)
- The resident size (the size of the part of the address space that's held in memory) of the process address space (RSS)
- The total shared address space
- The total private address space

There are many good tools for examining memory usage and estimating the amount of memory required for a certain process.

Solaris tools

One common tool is `/usr/ucb/ps uax`. Here's an example of the sort of data it gives:

```
% /usr/ucb/ps uax
USER       PID %CPU %MEM   SZ  RSS TT       S    START  TIME COMMAND
root     16755  0.1  1.0 1448 1208 pts/0    O 17:33:35  0:00 /usr/ucb/ps uax
root         3  0.1  0.0    0    0 ?        S   May 24  6:19 fsflush
root         1  0.1  0.6 2232  680 ?        S   May 24  3:10 /etc/init -
root       167  0.1  1.3 3288 1536 ?        S   May 24  1:04 /usr/sbin/syslogd
root         0  0.0  0.0    0    0 ?        T   May 24  0:16 sched
root         2  0.0  0.0    0    0 ?        S   May 24  0:00 pageout
gdm      14485  0.0  0.9 1424 1088 pts/0    S 16:17:57  0:00 -csh
```

Note that kernel daemons like `fsflush`, `sched`, and `pageout` report SZ and RSS values of zero; they run entirely in kernel space, and don't consume memory that could be used for running other applications. However, `ps` doesn't tell you anything about the amount of private and shared memory required by the process, which is what you really want to know.

The newer and much more informative way to determine a process's memory use under Solaris is to use `/usr/proc/bin/pmap -x` *process-id*. (In Solaris 8, this command moved to */usr/bin*.) Continuing from the previous example, here's a `csh` process:

```
% pmap -x 14485
14485:  -csh
Address   Kbytes Resident Shared Private Permissions       Mapped File
00010000     144     144      8     136 read/exec         csh
00042000      16      16      -      16 read/write/exec   csh
00046000     136     112      -     112 read/write/exec    [ heap ]
FF200000     648     608    536      72 read/exec         libc.so.1
FF2B0000      40      40      -      40 read/write/exec   libc.so.1
FF300000      16      16     16       - read/exec         libc_psr.so.1
FF320000       8       8      -       8 read/exec         libmapmalloc.so.1
FF330000       8       8      -       8 read/write/exec   libmapmalloc.so.1
FF340000     168     136      -     136 read/exec         libcurses.so.1
FF378000      40      40      -      40 read/write/exec   libcurses.so.1
FF382000       8       -      -       - read/write/exec    [ anon ]
FF390000       8       8      8       - read/exec         libdl.so.1
FF3A0000       8       8      -       8 read/write/exec    [ anon ]
FF3B0000     120     120    120       - read/exec         ld.so.1
FF3DC000       8       8      -       8 read/write/exec   ld.so.1
FFBE4000      48      48      -      48 read/write         [ stack ]
--------  ------  ------  ------  ------
total Kb    1424    1320     688     632
```

This gives a breakdown of each segment, as well as the process totals.

Linux Tools

The `ps` command under Linux works essentially the same way it does under Solaris. Here's an example (I've added the header line for ease of identifying the individual columns):

```
% ps uax | grep gdm
USER       PID %CPU %MEM  SIZE   RSS TTY STAT START   TIME COMMAND
gdm      12329  0.0  0.7  1548   984  p3 S   18:18   0:00 -csh
gdm      13406  0.0  0.3   856   512  p3 R   18:37   0:00 ps uax
gdm      13407  0.0  0.2   844   344  p3 S   18:37   0:00 grep gdm
```

While `ps` is ubiquitous, it's not very informative. The Linux equivalent of Solaris's `pmap` are the entries in the */proc* filesystem, which tell you very useful things.

One of the nicest features in Linux is the ability to work with the */proc* filesystem directly. This access gives you good data on the memory use of a specific process, akin to `pmap` under Solaris. Every process has a directory under its process ID in the */proc* filesystem; inside that directory is a file called *status*. Here's an example of what it contains:

```
% cat /proc/12329/status
Name:   csh
State:  S (sleeping)
Pid:    12329
PPid:   12327
Uid:    563     563     563     563
Gid:    538     538     538     538
Groups: 538 100
VmSize:     1548 kB
VmLck:         0 kB
VmRSS:       984 kB
VmData:      296 kB
VmStk:        40 kB
VmExe:       244 kB
VmLib:       744 kB
SigPnd: 0000000000000000
SigBlk: 0000000000000002
SigIgn: 0000000000384004
SigCgt: 0000000009812003
CapInh: 00000000fffffeff
CapPrm: 0000000000000000
CapEff: 0000000000000000
```

As you can see, this is a lot of data. Here's a quick Perl script to siphon out the most useful parts:

```
#!/usr/bin/perl
```

```
$pid = $ARGV[0];
@statusArray = split (/\s+/, `grep Vm /proc/$pid/status`);
print "Status of process $pid\n\r";
print "Process total size:\t$statusArray[1]\tKB\n\r";
print "            Locked:\t$statusArray[4]\tKB\n\r";
print "      Resident set:\t$statusArray[7]\tKB\n\r";
print "              Data:\t$statusArray[10]\tKB\n\r";
print "             Stack:\t$statusArray[13]\tKB\n\r";
print " Executable (text):\t$statusArray[16]\tKB\n\r";
print "  Shared libraries:\t$statusArray[19]\tKB\n\r";
```

Concluding Thoughts

There's a simple moral to this story: your memory requirements depend heavily on how many users there are and what they are doing. Large technical applications such as computational fluid dynamics or protein structure calculations can require incredible amounts of memory. It is easy to imagine a computer with 2 GB of memory that could support only one such job, whereas that same system could support several hundred interactive users quite comfortably.

One of the greatest improvements in computing in the last two decades has been the ability to assemble large amounts of memory for relatively little cost. Take advantage of it.

Footnotes:

1. Paging and swapping are often used interchangeably. However, they mean very different things in practice, so we'll be careful not to mix them.

2. In some cases, such as where resistance to ionizing radiation is required, magnetic core memory is still used.

3. One exception is the Sun Ultra Enterprise hardware line, which will create 6-way interleaves.

4. This was a significant problem in the large pre-UltraSPARC systems, such as the SPARCcenter 2000E.

5. Often called a COW fault; not to be confused with a ruminant falling into a chasm.

6. By default, the very end of the first swap partition is used to store kernel crash dumps under Solaris 7.

7. This number is typically very large. On UltraSPARC-based Solaris systems, it is about 3.75 GB.

8. There are typically 100 clock ticks per second.

9. The algorithm was later backported to Solaris 2.6.

10. Keep in mind that having not enough slightly faster memory is probably worse than having enough slightly slower memory. Be careful.

11. This is done in order to ensure coherency between main memory and the caches.

12. Most notably, any separate instruction caches are not measured.

Back to: System Performance Tuning, 2nd Edition
