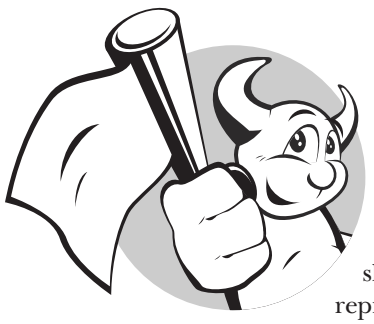
18

SYSTEM PERFORMANCE



"It's slow." That's one of the most dreaded phrases a systems administrator can hear. The user doesn't know why the system is slow—it just "feels" that way. Usually there's no test case, no set of reproducible steps, and nothing particularly wrong. These two words can cause the systems administrator hours of work, digging through the system trying to figure out what's going on.

One phrase is still more dreadful, especially after you've invested those hours debugging the problem: "It's still slow." For an inexperienced systems administrator, slow systems are easy to accelerate: Buy faster hardware. This generally fixes speed problems. It also costs a lot of money and simply conceals whatever's wrong, without really using the equipment you have.

FreeBSD includes many tools designed to help you examine system performance, and to give you the information you need to actually find out what's slowing things down. That will tell you what you need to do to fix the problem. You might very well need faster hardware, but you can quite possibly shift the load around within a system and improve overall performance. The first step is to understand what your problem really is.

Computer Resources

Speed problems are generally caused by running more on a computer than the computer can handle. That seems obvious, but think about it a moment. What does that mean?

A computer has four basic resources: disk input/output,[1] the network bandwidth, memory, and CPU. If any one of these is filled to capacity, the others cannot be used to their maximum effect. For example, your CPU might very well be waiting for a disk to deliver data or for memory to finish paging. A faster CPU won't increase system performance in this case.

Simply upgrading hardware when the system slows down does fix speed problems, but not in the way that you might think. If you have a program that fills up the system memory, buying a new system with a faster CPU will probably fix the problem. A new system probably has more memory than the old one, after all!

By identifying what the system is running short on, and addressing only that need, you can stretch your existing hardware much further. After all, why buy a whole new system when a couple hundred dollars of memory will fix the problem? (Of course, if your goal is to rotate this "slow" system into place as your new desktop, that's another matter.)

Perhaps you can reschedule work; one common cause of system slowdowns is running multiple large programs simultaneously. For example, I once scheduled a massive database log rotation that moved and compressed gigabytes of files at the same time as the system's automated daily checks. Since the job required shutting down the main database, and hence created system downtime, speed was crucial. Performance on both processes slowed unbearably. Rescheduling the log job greatly reduced downtime.

We're going to examine several FreeBSD tools for examining what a system is doing. Armed with that information, we'll consider how to fix some performance issues. We have separate tools to examine each of the potential bottlenecks.

FreeBSD changes continually, and later systems might have new tuning and performance features. Take a look at tuning(7) on your system to find any new performance tips. We'll cover tuning information that is useful on any FreeBSD (and almost any UNIX) system.

NOTE *One word you're going to keep stumbling across in this chapter is "abnormal." As the systems administrator, you're supposed to know what is normal for your system. It's somewhat like art: You might not be able to define normal, but you need to recognize it when you see it. It's a good idea for you to use these tools to check your systems regularly when they're behaving correctly, so you will have a good idea of what is out of whack when the system slows down. We'll also look at some long-term monitoring tools, so you can gauge system performance over months or years.*

[1] Technically, network bandwidth is part of input/output. However, it's special enough that we'll treat it separately.

Disk Input/Output

We looked at disk operations in some detail in Chapter 16. When it comes to performance, disk speed is usually a big bottleneck. If programs are waiting for disk activity to complete before proceeding, they will slow your system down. (This is commonly called "blocking on disk," meaning that the disk is blocking program activity.) The only real solution for this is to use a faster disk or a RAID array, or to split your disk activity between two disks.

How do you know if your disk is actually blocking program activity? We'll look at that in "Using Vmstat," later in the chapter.

Network Bandwidth

If your system performance slowdown is due to network problems, you need more bandwidth. In short: You can only push as much bandwidth as you have. If your T1 is full, you need more bandwidth. If your system cannot fill the existing bandwidth, use the tools discussed in Chapter 5 to increase system capacity.

To check for this problem, begin by monitoring how much bandwidth your system is using. Chapter 15 discusses how to generate long-term graphs of bandwidth usage. We also discussed networking in Chapter 5. Consult `netstat -m`, and increase your kernel's NMBCLUSTERS, as described in Chapter 4. That's really all there is to it.

Other system conditions are more complicated.

CPU and Memory

The top(1) tool is a good place to start if you're examining a system that seems to be running slowly. It provides a good overview of system status, but it only shows information about the CPU and memory usage; input/output and bandwidth are not touched.

Using Top

To read a top display, you must understand a great deal about how the system works, so we'll spend a good chunk of time on this. To run top, just type **`top`**. To display kernel processes as well as user programs, use `top -S`. You'll see a display much like the following, and it will refresh every few seconds.

```
# top -S
❶ last pid:   436;  ❷ load averages:  0.14,  0.08,  0.07    ❸ up 0+01:06:16 08:12:26
❹ 46 processes:  3 running, 43 sleeping
❺ CPU states:  1.2% user,  0.0% nice,  0.8% system,  0.0% interrupt, 98.1% idle
❻ Mem: 70M Active, 102M Inact, 26M Wired, 6016K Cache, 41M Buf, 107M Free
❼ Swap: 200M Total, 200M Free
❽
  PID USERNAME PRI NICE  SIZE    RES STATE    TIME   WCPU    CPU COMMAND
  287 mwlucas    2   5  2892K  2136K select   0:13  0.10%  0.10% xsysinfo
```

(continued on next page)

```
  378 mwlucas    2   0   101M 64920K RUN      0:08  0.10%  0.10% soffice.bin
  376 mwlucas    2   0 35372K 32736K RUN      0:13  0.05%  0.05% mozilla-bin
  274 mwlucas    2   0 28208K 26304K select   1:01  0.00%  0.00% XFree86
  170 root       2   0   912K   508K select   0:08  0.00%  0.00% moused
  277 mwlucas    2   0  3888K  3116K select   0:03  0.00%  0.00% wmaker
    5 root      18   0     0K     0K syncer   0:00  0.00%  0.00% syncer
  430 mwlucas   28   0  1912K  1160K RUN      0:00  0.00%  0.00% top
  399 mwlucas    2   0  4500K  4000K select   0:00  0.00%  0.00% Eterm
...
```

Very tightly packed, isn't it? Top tries to cram as much data as possible into a standard 80-character by 25-character terminal window. The display updates every two seconds, so you have a fairly accurate, close to real-time, view of your system. We'll take this a piece at a time and explain what every entry means.

PID Values

Every process on a UNIX machine has a unique process ID or PID. Whenever a new process is started, it is assigned a PID one greater than the previous process. The `last pid` value is the last process ID used in the system. In the previous example, the `last pid` is 436 (❶). The next process to be created will be 437, then 438, and so on. You can watch this increment to see if an abnormal number of processes is being created. Hopefully, you've looked at your system to see how quickly this number rises when things are running well. If the `last pid` value keeps climbing rapidly, programs are being started and stopped very quickly. This might indicate some daemon that keeps crashing, or a user trying to start too many programs.[2]

Load Average

The `load average` (❷) is a somewhat vague number that's intended to give a rough impression of the amount of load on the system.[2] The load average equals the average number of processes waiting for CPU time, plus the average number of jobs that are waiting for access to the disk. An acceptable load average depends on the system; if the numbers are abnormally high, you should investigate. Many 486s feel bogged down at a load average of 3, while some modern systems feel snappy at a load average of 10.

Top lists three load averages. The first (0.14 in our example) is the load average over the last minute, the second (0.08) is for the last 5 minutes, while the last (0.07) is for the previous 15 minutes. If your 15-minute load average is high, but the 1-minute load is low, you had a major spike in activity that has passed. How well did your system hold up? On the other hand, if the 15-minute value is low, but the 1-minute average is high, something happened within the last 60 seconds and may still be going on now. If all of the load averages are high, the condition has persisted for the whole 15 minutes.

[2] Some users actually try to use up system resources by starting programs. This is called a *forkbomb*. These users are like script kiddies, but not as educated.

Uptime

The last entry on the first line is the *uptime* (❸), or how long the system has been running. The system in our example has been up for one hour and six minutes, and the current time is 08:12:26. I'll leave it up to you to figure out when the system booted.

Process Counts

On the second line you'll find information about processes that are currently running on the system (❹). Running processes are actually doing work; they're answering user requests, handling mail, or whatever else is going on. Sleeping processes are waiting for input from one source or another, and are just fine. Processes in other states are usually waiting for a resource to become available, or are hung in some way. Large numbers of nonsleeping, nonrunnable processes can be a hint of trouble. Investigate further to find out which processes those are.

Process Types

The `CPU states` line (❺) indicates what percentage of available time the CPU spends handling different types of processes and other duties. It shows five different process types: `user`, `nice`, `system`, `interrupt`, and `idle`.

User processes are average everyday programs; they could be daemons run by root, commands run by regular users, or whatever. If it shows up on the list of system processes (that is, on `ps -ax`), it's a user process.

Nice processes are those whose priority has been deliberately manipulated by the user. We'll look at this in some detail in "Reprioritizing with Niceness."

System processes are in the kernel, and they include things such as virtual memory handlers, running networking, writing to the disk, and so on.

The *interrupt* category shows how much time the system spends handling interrupt requests (IRQs).

Lastly, the *idle process* shows how much time the system spends doing absolutely nothing. If your system CPU regularly has a very low idle time, you might want to start thinking about rescheduling some jobs or getting a faster processor.

NOTE *When you're working on a multi-CPU system, keep in mind that top displays the average usage among all the processors. You might have one processor completely tied up compiling something, but if the other processor is idle, top will show only 50 percent usage.*

Memory

Then we have the `Mem` line, representing actual physical RAM (❻). Unlike Windows, which simply divides memory into "used" and "unused" categories, FreeBSD uses memory in several different ways.

Active memory is the total amount of memory in use at the moment for running user programs and their data. When a program ends, the program information is put into *inactive* memory and the data pulled from the disk is put in the *cache* memory.

Similarly, the *Buf* entry shows the size of the memory buffer. The memory buffer contains data recently called from disk.

Free memory is unused.

Wired memory is memory used for in-kernel data structures, as well as for particular system calls that must have a particular piece of memory immediately available. Wired memory is never swapped out.

Swap

Then we have the `Swap` line, (❼), which simply represents the total swap available and how much is in use. *Swapping* is when the system uses the disk drive as additional memory. We'll look at swap in more detail later in the chapter.

Process List

Finally, we have a list of the processes on the system and their basic characteristics (❽). The table format is designed to present as much information as possible in as little space as possible. Every process is on its own line. Let's look at each column in the following sections.

PID First we have the process ID number, or PID. Every process running on the system has a unique PID. When you issue kill commands, you use the PID to identify the process you want to affect.

Username Next is the username of the person running the process. If multiple processes consume large chunks of CPU or memory, and they are all owned by the same user ID, you know who to talk to.

Priority and Nice The PRI (priority) and NICE columns are interrelated, and indicate how much precedence the system gives these processes. We'll talk about priority and niceness a little later in the chapter.

Size Size is the amount of memory that the system has set aside for this process.

Resident Memory The RES column shows how much of the program is actually in memory, or resident, at the moment. A program might have a huge amount of memory reserved for it, but only be using a small fraction of it.

State The STATE column shows what the process is doing at the moment. Processes can be in a variety of states at any given time: waiting for input, sleeping until something wakes them, actively running, and so on.

Time The TIME column gives the total length of time that the process has been running.

CPU Usage The WCPU column gives a weighted CPU usage that shows the percentage of CPU time that the process is using, as adjusted for the process's priority and niceness. The CPU column shows what percentage of CPU time the program is actually using.

Command Name Finally, in the COMMAND column, we have the program name.

Memory Usage

If your system is running slowly, check its memory and CPU usage first. While they're no more likely to be running amok than any other part of the system, they're the easiest to measure. Let's discuss memory first.

FreeBSD errs on the side of caching recently accessed data because a surprising amount of information is read from disk time and time again. If this information can be cached in physical memory, it can be accessed very quickly. If the system needs more memory, it dumps the oldest cached chunks in favor of new data.

For example, the example top output we're discussing is from my laptop, which is using a lot of buffer and inactive memory. Part of that is due to my Web browser. I started Mozilla when I booted the system yesterday morning so I could check my morning comics.[3] For a couple of moments, the disk light stayed solidly lit while the system read the program off the disk. I then shut the browser down so I could do some work.

Since this Web browser was accessed, it sat in the system buffer cache. When I started the browser again this morning, it only had to be called out of cache rather than from disk, so it started much more quickly. Had I started some other large process, it would have dumped that Web browser from the cache to read in more data.

If your system is operating well, you will have at least a few megs of free memory. If you have more than a few megs free, your system is not being used to nearly its full potential. In the example earlier, I could get rid of 128MB of RAM and not affect system performance much at all.

If you have a good chunk of memory in cache or buffer, you don't have a memory shortage. You might make good use of more memory, but it isn't strictly necessary. Similarly, if you have a lot of free memory, you probably don't have a memory shortage. If active and wired memory is consuming most of your available memory, more RAM wouldn't hurt.

When you're out of free space, and have little or no memory in cache or buffer, you should investigate your memory use further. You may well have a memory shortage. Take a look at the "Using Vmstat" section later in the chapter to check.

Swap Space Usage

Virtual memory, or *swap* helps cover brief RAM shortages. For example, if you're untarring a huge file, you might easily fill up all your physical memory and have to start using virtual memory. It's not worth buying more RAM for this occasional use when swap suffices.

Like memory cache, swap caches data that it has handled recently, and once you've touched swap, it never returns to being free. For example, I have a server that has been up for 772 days at this writing. At one point, I used about a hundred megs of swap to handle a massive compile. My top display still shows that I'm using that 100MB of swap, while I have over 200MB of memory free.

[3] Sluggy Freelance (www.sluggy.com) and Help Desk (www.ubersoft.net), if anyone cares.

Using swap space is not a bad thing, especially since a program will typically spend 80 percent of its time running 20 percent of its code. Since much of the rest of that time spent running is startup, shutdown, and error code, you can safely let those bits go to swap space and have minimal impact.

So don't worry if you find that you're using a bit of swap space on occasion. But, if you're constantly using swap, you probably need more memory.

CPU Usage

A processor can do only so many things a second, and if you want to do more than your CPU can handle, the requests will start to queue up. You'll develop a processor backlog, and the system will slow down. That's CPU usage in a nutshell.

If top shows your CPU hovering around 100 percent all the time, you must take action. While new hardware is certainly an option, you do have other choices. For one, investigate the processes running on your system to see whether they're all necessary. Did some junior administrator install the SETI@Home client (/usr/ports/astro/setiathome) to hunt for aliens with spare CPU cycles? Are there things running that were important at one time, but are now unnecessary? Find and kill those unnecessary processes and make sure that they won't start the next time the system boots.

Once that's done, evaluate your system performance again. If you still have a problem, try rescheduling or reprioritizing.

Rescheduling

Rescheduling is easier than reprioritizing, and it is a relatively simple way to balance system processes so that they don't load up on CPU time. As discussed in Chapter 9, you can use cron(1) to schedule system tasks for various times, but users can use it too. If you have users who are running massive compile jobs or doing huge database queries, you might consider using cron to schedule them to run at night. Frequently, jobs such as the monthly billing database search can be run between 6 PM and 6 AM, and nobody will care. Similarly, you could schedule your `make buildworld && make buildkernel` to start at 1 AM.

Reprioritizing with Niceness

If rescheduling won't work, you're left with reprioritizing, which is a bit trickier. When reprioritizing, you tell UNIX to change the importance of a given process.

For example, if you want a software install to run, but only when nothing more important is running, you reprioritize it with "niceness," which is simply a relative measure of how much CPU time a process demands. The nicer a process is, the less CPU time it demands. The default niceness is 0, but niceness runs from 20 (very nice) to -20 (not nice at all). (This might seem backwards; you could argue that a higher number should mean a higher priority. That leads to a language problem, though, as calling this factor "crankiness" or "greed" didn't seem like a good idea at the time.)[4]

[4] This might be one of the few circumstances where common sense won out in naming UNIX commands.

In the top display seen earlier (in the "Using Top" section) you saw a PRI column for process priority. FreeBSD calculates a process priority by combining a variety of factors, including niceness, and runs high-priority processes first whenever possible. Niceness affects priority, but you cannot directly edit priority.

If you know that your system is running low on CPU capacity, you can choose to start a command with nice(1) to assign the command a priority. Specify the desired niceness level by putting a single dash in front of the command. For example, to start a `make buildworld` at nice 15, you would run this command:

```
# cd /usr/src
# nice -15 make buildworld
```

Only root can assign a negative niceness to a program. To run a program with negative niceness, use a double dash (`nice --5`). For example, if you have a critical kernel patch that must be applied as soon as possible, and you want the compile to finish as quickly as possible, use a negative niceness like so:

```
# cd /sys/i386/compile/MYKERNEL
# nice --20 make depend && nice --20 make all install
```

Usually, you won't have the luxury of telling a command to start off nicely, but will instead need to change a process's niceness on the fly (generally, when you find out that it's soaking up all your CPU). You can do so with renice(8), which will reprioritize by process ID or owner. To change the niceness of a process ID, you run renice with the new niceness and the process ID.

For example, one of my systems has a FreeBSD CVSup mirror. If I find that the mirror is taking up so much CPU time that it's getting in the way of things I have to do, I can change its niceness to 20. The maximum niceness we can use is 20, which basically tells the system to run this command only if nothing else at all wants CPU time. To renice a running process, I first need to know its process ID. I know the process is named cvsupd because I've looked at this system's top output over the last several months. I then look at all the processes running on the system, and pull out the one for cvsupd with the following command:

```
# ps -ax | grep cvsupd
  322  ??  Is     0:00.01 /usr/local/sbin/cvsupd -C 5 -b /test2 -s sup
#
```

The first column in the preceding ps output is the process ID, PID 322. Now to renice it, I would enter the following:

```
# renice 20 322
322: old priority 0, new priority 20
#
```

Boom! The cvsupd daemon will now only run when nothing else requiring system time is running. This will greatly annoy users of the service, of course, but I presumably have a good reason for doing so. (Since this is a private mirror, not a public one, I feel no particular need to be kind to my users.)

To renice every process owned by a user, use the -u flag. For example, to make my processes more important than anyone else's, I could enter this command:

```
# renice -5 -u mwlucas
1000: old priority 0, new priority -5
#
```

The 1000 is my user ID number on this system. Again, presumably I have a very good reason for doing this besides a need for personal power.[5]

NOTE *Renicing, rescheduling, and process management don't create additional CPU time, they simply rearrange the CPU time you do have. If you cannot reschedule processes, and you cannot satisfactorily renice things to tune the way the system behaves, you really do need faster or additional hardware. Some systems have an extra motherboard slot for an additional CPU, which is a quick and inexpensive way to boost performance when the system is CPU-bound. If you have multiple CPUs, definitely take a look at the discussion of SMP in Chapter 11.*

When Swap Goes Bad

I said earlier that using swap space isn't bad in and of itself because swap space is used as virtual memory. (In other words, memory space on the hard drive is being used in the same way as RAM.) Swap space is much slower than chip memory, but it does work in a pinch, and many programs don't need to have everything in RAM in order for them to run. If programs spend 80 percent of their time in 20 percent of their code, then 80 percent of their bulk can be put into swap space without seriously impacting performance.

Many sysadmins use the term swapping generically, lumping two different activities (paging and swapping) together without understanding the crucial difference between them.

Paging

When you read about virtual memory, you'll see references to *pages*. A page is simply a section of memory, 4KB on x86 hardware under FreeBSD. (Different platforms have different page sizes.)

Data moves between real and virtual memory in units of pages. *Paging* happens when a portion of a running program is moved onto swap. This process can actually improve performance on a heavily loaded system because unused bits can be stored on disk until they're needed.

[5] Being a selfish person doesn't qualify as a good reason. Or so I've been told.

Swapping

Swapping describes what happens when an entire runnable process is moved into swap. If the computer doesn't have enough physical memory to store a process that isn't being run at that particular microsecond, the system can move the entire process to swap. Then, the next time the CPU runs that process, the process's memory is moved from swap into physical memory, and some other process is probably consigned to swap.

The problem with swapping is that disk usage goes through the roof and performance drops drastically. Since requests take longer to handle, there are more requests of the system at any one time. And logging in to check the problem only makes the situation worse, because logging runs an extra system process. This performance hit is sometimes called the *death spiral*.

Memory shortages will hurt system performance more than anything else. If you're frequently swapping, you *must* get more memory or resign yourself to lousy[6] performance.

NOTE *Every system has bottlenecks, or places where performance is limited. If you eliminate one bottleneck, performance will increase until another bottleneck is hit. The system will work at the fastest speed allowed by the slowest component in the system, also called bounds. For example, a Web server is frequently network-bound because the slowest part of the system is the Internet connection. If you upgrade the Internet connection, the system will hand out Web pages as fast as either its CPU or disk allows.*

Are You Swapping or Paging?

FreeBSD includes several programs for examining system performance. Among those are vmstat(8), iostat(8), and systat(1). We'll discuss vmstat because I find it to be the most helpful. Iostat is similar to vmstat, and systat provides similar information in a more graphic format.

Using Vmstat

Vmstat(8) shows virtual memory statistics at the current time. While its output takes some getting used to, it is very good at showing large amounts of data in a very small space. Type **vmstat** at the command prompt, and follow along.

```
# vmstat
 procs      memory      page                    disks     faults      cpu
 r b w     avm    fre  flt  re  pi  po  fr  sr ad4 da0   in   sy  cs us sy id
 0 0 0    7096 479140   21   0   0   0   9   0   0   0  331  102 437  0  1 99
#
```

The display is divided into six sections: process (procs), memory, paging (page), disks, faults, and cpu. We'll look at each then quickly and then dive into detail on the bits that are most important for investigating your performance issues.

[6] I would use a better word than "lousy," but my editor frowns upon the flavorful language I learned from an ex-sailor co-worker.

Processes

There are three columns under the procs heading.

`r` Lists the number of processes that are waiting to run on the CPU. These are processes that are ready to run, but which simply cannot get access to the CPU to execute. If this number is high, your CPU is bottlenecking your system.

`b` Gives the number of processes that are blocked waiting for system input or output—generally, waiting for disk access. These processes will run as soon as they get their input. If this number is high, your disk is the bottleneck.

`w` Shows processes that are runnable but are entirely swapped out. If you start having processes swapped out on a regular basis, your memory is inadequate for the work you are doing on the system.

Memory

The memory section has two columns.

`avm` Shows the average number of pages of virtual memory that are in use. If this value is abnormally high or increasing, your system is using up virtual memory.

`fre` Shows the number of pages that remain available for use. If this value is abnormally low, you have a memory problem.

Paging

The paging section shows how hard the virtual memory system is working.

`flt` Shows the number of page faults, where the information needed is not in memory and needs to be fetched from the disk.

`re` Shows how many pages have been reclaimed or reused from cache.

`pi` Short for *pages in*, it shows how many pages are moving from physical memory to swap.

`po` Short for *pages out*, it shows how many pages are moving from swap to real memory.

`fr` and `sr` Show how many pages are freed and scanned per second, respectively. You don't have to worry about these too often, unless your system is under very heavy memory load.

Disks

The disks section shows each of your disks by device name. The number shown is the number of disk operations per second. You should divide your disk operations between different disks whenever possible, and arrange them on different buses (as discussed in Chapters 1 and 13). If one disk is obviously busier than the others, and the system has operations waiting for disk access, consider moving some frequently used files from one disk to another.

Faults

The faults section shows system faults. Faults, in this case, aren't bad, they're just received system traps and interrupts.

`in` Shows the number of system interrupts (IRQ requests) the system received in the last five seconds.

`sy` Shows the number of system calls in the last five seconds.

`cs` Gives the number of *context switches*, or times the CPU changed from doing one thing to doing another.

CPU

Finally, the CPU section shows how much time the system spent doing user tasks (`us`), and system tasks (`sy`), and how much time it was idle (`id`). This is the same information presented by top.

Making Use of vmstat Information

So, how do you use this information? First, check the first three columns to see what the system is waiting for when it's slow. If you're waiting for CPU access (the `r` column), then you're short on CPU horsepower. If you're waiting for disk activity (the `b` column), then your disks are your bottleneck. If you're swapping (the `w` column), then you're short on memory. Simple enough, eh?

If you're having problems with memory, you can expect the `page` section to have very high values. (The details of virtual memory management are an arcane science that I won't cover in depth here.) They key is to know what your system normally looks like, and hence what would be abnormal.

Monitoring Multiple Disks

Vmstat shows what's happening on your disks and where data is being written. The number of disk operations per second is a valuable clue to how well your disks are handling their load.

However, if you have a lot of disks, you may notice that they don't all appear on the vmstat display. Vmstat is biased toward fitting into an 80-column display, and hence cannot list every possible disk on the system. If you don't mind overflowing 80 columns, you can use vmstat's `-n` flag to set the number of drives you want to display. The 80-column limit is important on a system console, but it can easily be overcome when you're using SSH from a workstation.

Continuous Vmstat

When using vmstat, you're probably more interested in what's happening over a period of time than in taking a brief snapshot. Use vmstat with the `-w` flag to run it as a continuously updating display and to specify the number of seconds between updates. Many internal system counters are recalculated every five seconds, so five seconds is the minimum recommended time between updates.

```
# vmstat -w 5
 procs      memory      page                    disks     faults      cpu
 r b w     avm    fre  flt  re  pi  po  fr  sr ad0 md0   in   sy  cs us sy id
 1 0 0  165208  51408  431   0   0   0 408   4   0   0  243 2656 255 13  3 83
 0 0 0  165208  51408    8   0   0   0   0   0   0   0  267  829 232  0  2 97
 1 0 0  172480  51408    9   0   0   0   2   0   0   0  277  986 279  2  1 97
 1 0 0  174584  51108   44   0   0   0  21   0   0   0  262 3694 269  1  3 96
...
```

Press **CONTROL-C** when you're done, and just sit and watch your system do its work, and see how it reacts when scheduled jobs kick off. In the preceding example, we have the occasional moment where processes are waiting on CPU time (as shown by the intermittent `1` in the `r` column), but the disk and memory all seem to be behaving well. An occasional wait for some resource doesn't mean that you need to upgrade that system component; if performance is acceptable, don't worry about it.

Real-World Performance Tuning

All this theoretical stuff is nice, but how do you troubleshoot performance in real life? At this point, I hope you've actually read the previous chapters of this book, as we're going to be referring to information brought up in all sorts of different places. We'll use a real-world test on a real-world system to demonstrate how performance troubleshooting works.

The standard FreeBSD torture test is the `make world` process run during an upgrade from source. It pounds on the CPU and the disk, and absorbs all the memory it can get its greedy hands on. We'll focus on the `make buildkernel` stage of this process because it's shorter than the `make buildworld` stage, which makes for a better test. Let's see how we can use the techniques and information presented here to reduce the total time needed to run the build.

The system I'll be using for the test has two 1 GHz Pentium CPUs, a new 60GB IDE drive, two somewhat elderly 4.5GB SCSI-2 drives, and 512MB of RAM. It's running a few smaller programs as well, but there is no X server, window manager, or Web server. Initially, the system is installed on the one IDE disk; the SCSI disks are completely idle. The install is fairly default, with soft updates (see Chapter 16) set on the /usr partition.

Fairness in Benchmarking

Benchmarking is a difficult task. We're not officially benchmarking here, but we'll still do some things to make sure that each run of our test is as fair as possible. I'll reboot the system between tests to eliminate anything that might be lurking in the buffer cache. In this case, we want to improve performance ourselves, not use FreeBSD's buffering and caching to do it for us. Similarly, I'll remove /usr/obj (where the buildkernel creates its files) between runs.

NOTE *While* `make buildkernel` *is a fairly standard sort of test, don't assume that it is the be-all and end-all of FreeBSD performance. I'm using it here because it's a standard process, and everyone has access to it. Test performance on your systems using programs and commands that you actually use, not arbitrary benchmarks.*

The Initial Test

To begin, we'll record datestamps from the beginning and end of each run. This will give us an absolute measurement of how any changes affect performance:

```
# date >> timestamps && make buildkernel && date >> timestamps
```

This command records the start and stop times in a file called timestamps, and runs `make buildkernel`.

Now start the build and look at top, the first few lines of which are shown here:

```
last pid:  6262;  load averages:  0.87,  0.37,  0.15    up 0+01:00:43  12:14:17
46 processes:  2 running, 44 sleeping
CPU states: 21.2% user,  0.0% nice, 29.4% system,  0.6% interrupt, ❶ 48.8% idle
Mem: 16M Active, 38M Inact, 36M Wired, 2240K Cache, 61M Buf, ❷ 407M Free
Swap: 2048M Total, 2048M Free

  PID USERNAME PRI NICE   SIZE    RES STATE  C   TIME   WCPU    CPU COMMAND
  529 root      96    0  2420K  1956K ect    0   0:01  0.10%  0.10% sshd
  275 root      96    0  1020K   588K ect    0   0:08  0.00%  0.00% moused
  354 root      96    0  2420K  1956K ect    1   0:00  0.00%  0.00% sshd
  223 root       8    0  1084K   688K slp    0   0:00  0.00%  0.00% diskcheckd
  252 root      96    0  2364K  1676K ect    1   0:00  0.00%  0.00% sshd
...
```

You can see right away that the system is not short on memory; with 407MB free (❷) it should be good for quite some time. The CPU is 48.8 percent idle (❶), so a lack of processor time does not appear to be the bottleneck.

Let's look at a snippet of vmstat output, updated every five seconds:

```
# vmstat 5
 procs      memory      page                    disks     faults      cpu
 r b w     avm    fre  flt  re  pi  po  fr  sr ad4 da0   in   sy  cs us sy id
 0 2 0   17952 396524  173   0   0   0 199   0   0   0  354  385 630  2  2 96
 2 1 0   18872 394908 2260   0   0   0 2281  0  31   0  369 2682 2342 37 16 47
 2 2 0   19268 394384 1801   0   0   0 1856  0   2   0  336 2107 1687 40 12 48
 0 2 0   19164 393768 2074   0   0   0 2143  0  15   0  353 2617 2162 32 14 53
 2 2 0   21680 389032 2045   0   0   0 1892  0   1   0  337 2349 1908 40 12 47
 0 2 0   16096 393452 1916   0   0   0 2242  0   1   0  338 2281 2240 39 14 47
 2 2 0   17888 390616 2260   0   0   0 2236  0   1   0  342 2830 2844 35 17 47
 1 2 0   18880 389728 2260   0   0   0 2337  0  30   0  370 2804 2909 35 19 46
 0 2 0   16484 391684 2031   0   0   0 2234  0   1   0  338 2477 2183 37 16 47
 1 2 0   18416 389052 2230   0   0   0 2219  0  11   0  352 2886 2876 33 18 49
...
```

Okay, something is definitely not correct here. The `r` column shows how many processes can be run but that can't get CPU time. Our system is almost 50 percent idle, yet some processes cannot get CPU time! What's going on?

Well, this is a multiple-processor system. Remember, a CPU does only one thing at a time. What we see here is that one CPU at a time is actually full, while the other isn't doing anything at all. The solution is to split the load between our CPUs. We can do this in a make with the `-j` flag, as discussed in Chapter 6. That'll be our next test.

When the `make buildkernel` finishes, take a look at the times to set our benchmark:

```
#more timestamps
Sun Aug 19 12:11:47 EDT 2001
Sun Aug 19 12:23:43 EDT 2001
#
```

Just 4 seconds under 14 minutes, or 716 seconds. That's our benchmark; can we beat it?

Using Both CPUs

Let's try using *both* our processors to see what happens to the time.

```
# date >> timestamps && make -j2 buildkernel && date >> timestamps
```

We could use numbers over 2 for `-j`, but 2 is a good place to start. In theory, this should use much more of our CPU.

Let's see how theory compares to reality:

```
last pid:  3855;  load averages:  1.08,  0.36,  0.16    up 0+00:07:18 12:36:45
51 processes:  1 running, 47 sleeping, 1 zombie, 2 mutex
CPU states: 23.0% user,  0.0% nice, 41.2% system,  0.0% interrupt, ❶ 35.8% idle
Mem: 16M Active, 14M Inact, 29M Wired, 8K Cache, 51M Buf, 442M Free
Swap: 2048M Total, 2048M Free

  PID USERNAME PRI NICE   SIZE    RES STATE  C   TIME   WCPU    CPU COMMAND
 2016 root       8    0   612K   512K t      1   0:01  1.46%  1.22% cc
  282 root      96    0  1020K   588K ect    1   0:01  0.00%  0.00% moused
 1653 root      96    0  1756K  1632K ect    1   0:01  0.00%  0.00% make
  351 root      96    0  2420K  1952K ect    1   0:01  0.00%  0.00% sshd
...
```

Only 35.8 percent idle is a lot better (❶); both processors are working now. Let's check vmstat:

```
 procs      memory      page                    disks     faults      cpu
 r b w     avm    fre  flt  re  pi  po  fr  sr ad4 da0   in   sy  cs us sy id
 0 4 0   19236 385696 2521   0   0   0 3021   0   8   0  350 3691 5099 52 27 21
 0 5 0   29304 374976 2768   0   0   0 2381   0  14   0  357 3929 5403 56 28 16
 0 2 0   20992 381444 3140   0   0   0 3634   0  25   0  364 4143 6585 52 32 16
 0 1 0   19348 378600 2925   0   0   0 3064   0  39   0  386 4542 8170 39 34 27
 0 6 0   25296 372936 3464   0   0   0 3356   0   7   0  349 4734 9110 47 37 16
 0 2 0   20456 374884 2584   0   0   0 2860   0  44   0  383 3683 5209 52 27 21
 0 6 0   27140 367312 2828   0   0   0 2660   0  10   0  352 4081 6572 52 32 16
...
```

Well, this is better on the CPU front. If you watch long enough, you'll see an occasional bubble of CPU shortage, but it's much better than it was before. (Momentary shortages are perfectly natural—you only need to worry when they keep recurring.) But the contents of the `b` column, which lists processes that cannot run because they're waiting for the disk, is alarming. It is never 0, which tells us that our process has become disk-bound.

Still, let's look at our current timestamps:

```
Sun Aug 19 12:35:25 EDT 2001
Sun Aug 19 12:46:46 EDT 2001
```

It's 11 minutes, 21 seconds, or 681 seconds. Using both CPUs chopped about 30 seconds off the process. While that's only about a 5 percent increase in speed, we got that increase without spending a dime on additional hardware. Our next problem is to reduce the I/O bottleneck.

Directory Caching

The least intrusive way to reduce disk input/output is to enable directory caching with the `vfs.vmiodirenable` sysctl (Chapter 16). We could do this without rebooting the system with sysctl(8) (Chapter 4). (You might already have this enabled, depending on your version of FreeBSD, but you should check it.)

```
# sysctl vfs.vmiodirenable=1
vfs.vmiodirenable: 0 -> 1
#
```

Since our test says we need to reboot the system anyway, though, let's set this sysctl in /etc/sysctl.conf, delete /usr/obj, and reboot. Next, we'll run the `make buildworld` with the same command. Our top output looks almost identical to the last run, so I won't bother showing it here again, and when we run vmstat, it also looks very similar. Here are the timestamps:

```
Sun Aug 19 13:21:58 EDT 2001
Sun Aug 19 13:33:15 EDT 2001
```

The total is 677 seconds. We've saved 4 whole seconds by caching directory lists. Why so little?

Well, the purpose of caching something is so it can be reused later. When you build a piece of software, the build process visits each directory just once. If you never return to the same spot and actually use the cache, it's pointless. We've seen a minor savings from the rare occasions when `make buildkernel` visits a directory repeatedly, but that's all. You'd see better improvements on processes such as a Web server, which accesses the same files over and over again.

So, our disk is still the bottleneck. It's time for some major surgery.

Moving /usr/obj

The `make buildkernel` process reads files under /usr/src and writes them under /usr/obj. Let's take one of our ancient SCSI disks and mount it on /usr/obj, leaving /usr/src on the new IDE disk. For our first test, we'll use a default mount without soft updates, just to illustrate a point. Our system disks now look like this:

```
# df
Filesystem  1K-blocks     Used    Avail Capacity  Mounted on
/dev/ad4s1a    248111    74081   154182    32%    /
/dev/ad4s1f   2032839   133492  1736720     7%    /test1
/dev/ad4s1g   2032839  1266476   603736    68%    /test2
/dev/ad4s1h  29497862  3842891 23295143    14%    /usr
/dev/ad4s1e   3048830   977220  1827704    35%    /var
procfs              4        4        0   100%    /proc
/dev/da0s1e   3525041        1  3243037     0%    /usr/obj
#
```

In theory, input and output will be spread between different disks. Our top output is similar, but vmstat looks different:

```
 procs      memory      page                    disks     faults      cpu
 r b w     avm    fre  flt  re  pi  po  fr  sr ad4 da0   in   sy  cs us sy id
 0 4 0   29200 329436 2433   0   0   0 2109   0   2  12  367 3390 3220 58 24 18
 0 0 0   15388 338376 2298   0   0   0 2939   0   1  23  390 3315 4487 49 24 26
 0 0 0   19124 336604 2453   0   0   0 2559   0   3  30  413 3672 5333 39 29 32
 0 3 0   23680 330252 2000   0   0   0 1818   0  14  65  489 2979 4874 29 24 46
 0 4 0   22832 329136 2628   0   0   0 2768   0   1  16  374 3783 5158 50 28 23
 0 5 0   23404 326976 2624   0   0   0 2702   0   0  15  373 3815 5550 52 29 20
...
```

We're still blocking, waiting for disk throughput, but take a look at the ad4 (IDE disk) and da0 (SCSI disk) columns. Load is now split between the two. When we're done, our timestamps look like this:

```
Sun Aug 19 13:59:54 EDT 2001
Sun Aug 19 14:11:41 EDT 2001
```

Seven hundred and seven seconds! That's just as bad as when we started! Ouch. A thing to remember, however, is that this decrepit SCSI drive, without soft updates, performed just as well as a modern IDE drive. Buying a modern SCSI drive would definitely enhance performance. Let's enable soft updates on our new /usr/obj and see what happens:

```
# umount /usr/obj
# tunefs -n enable /usr/obj
tunefs: soft updates set
# mount /usr/obj
#
```

Now delete everything in /usr/obj, reboot, and try again. A check of vmstat shows that disk throughput is unquestionably our bottleneck, again. With soft updates, our time goes down to 670 seconds. Soft updates gave us a total 6 percent improvement. While this certainly isn't great, the time savings add up over the course of a day or a long build.

You should now have a very good idea of how to tune your system. Play with it some more. Perhaps /usr/obj as a mirrored Vinum partition, with soft updates—this pulls the time down to 663 seconds, or a 7 percent improvement. That's about as good as it gets. Throughout it all, vmstat shows that disk throughput is the bottleneck.

Lessons Learned

In the preceding process we learned that disk speed is inarguably the bottleneck. This particular four-year-old SCSI disk handles data just as quickly as a modern IDE disk, but if we want faster performance we need a faster disk. Faster disks are much less expensive than a whole new machine, even if that new machine includes a faster disk.

Best of all, you can now go to your manager and say, "This is bad. We need a faster disk. Our vendor, AbsoluteBSD.com, has them for $400," and be certain of your facts. That's much better than saying, "This is bad; we need a new server."

Of course, programs other than `make buildkernel` have completely different requirements and must be evaluated separately. While a 5 to 10 percent increase isn't a huge performance boost, it can make the difference between doing maintenance during the normal maintenance window and pulling a desperate triple shift to get the new equipment slammed into place so that people can do their work the next morning.