there are no encryption arguments for *pam_unix.so* for the `auth` function. The manual pages also tell you nothing.

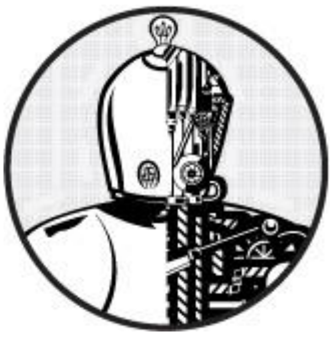
It turns out that (as of this writing) *pam_unix.so* simply tries to guess the algorithm, usually by asking the libcrypt library to do the dirty work of trying a whole bunch of things until something works or there's nothing left to try. Therefore, you normally don't have to worry about the verification encryption algorithm.

7.11 Looking Forward

We're now at about the midpoint in our progression through this book, having covered many of the vital building blocks of a Linux system. The discussion of logging and users on a Linux system has introduced you to what makes it possible to divide services and tasks into small, independent chunks that still know how to interact to a certain extent.

This chapter dealt almost exclusively with user space, and we now need to refine our view of user-space processes and the resources they consume. To do so, we'll go back into the kernel in Chapter 8.

Chapter 8. A Closer Look at Processes and Resource Utilization



This chapter takes you deeper into the relationships between processes, the kernel, and system resources. There are three basic kinds of hardware resources: CPU, memory, and I/O. Processes vie for these resources, and the kernel's job is to allocate resources fairly. The kernel itself is also a resource—a software resource that processes use to perform tasks such as creating new processes and communicating with other processes.

Many of the tools that you see in this chapter are often thought of as performance-monitoring tools. They're particularly helpful if your system is slowing to a crawl and you're trying to figure out why. However, you shouldn't get too distracted by performance; trying to optimize a system that's already working correctly is often a waste of time. Instead, concentrate on understanding what the tools actually measure, and you'll gain great insight into how the kernel works.

8.1 Tracking Processes

You learned how to use `ps` in 2.16 Listing and Manipulating Processes to list processes running on your system at a particular time. The `ps` command lists current processes, but it does little to tell you how processes change over time. Therefore, it won't really help you to determine which process is using too much CPU time or memory.

The `top` program is often more useful than `ps` because it displays the current system status as well as many of the fields in a `ps` listing, and it updates the display every second. Perhaps most important is that `top` shows the most active processes (that is, those currently taking up the most CPU time) at the top of its display.

You can send commands to `top` with keystrokes. These are some of the most important commands:

Spacebar	Updates the display immediately.
M	Sorts by current resident memory usage.
T	Sorts by total (cumulative) CPU usage.
P	Sorts by current CPU usage (the default).
u	Displays only one user's processes.
f	Selects different statistics to display.
?	Displays a usage summary for all `top` commands.

Two other utilities for Linux, similar to `top`, offer an enhanced set of views and features: `atop` and `htop`. Most of the extra features are available from other utilities. For example, `htop` has many of abilities of the lsof command described in the next section.

8.2 Finding Open Files with lsof

The lsof command lists open files and the processes using them. Because Unix places a lot of emphasis on files, lsof is among the most useful tools for finding trouble spots. But lsof doesn't stop at regular files—it can list network resources, dynamic libraries, pipes, and more.

8.2.1 Reading the lsof Output

Running lsof on the command line usually produces a tremendous amount of output. Below is a fragment of what you might see. This output includes open files from the init process as well as a running vi process:

```
$ lsof
COMMAND PID USER  FD  TYPE DEVICE     SIZE    NODE NAME
init      1 root cwd  DIR    8,1     4096       2 /
init      1 root rtd  DIR    8,1     4096       2 /
init       1 root mem   REG    8,      47040   9705817 /lib/i386-linux-
gnu/libnss_files-2.15.so
init       1 root mem   REG    8,1     42652   9705821 /lib/i386-linux-
gnu/libnss_nis-2.15.so
init       1 root mem   REG    8,1     92016   9705833 /lib/i386-linux-
gnu/libnsl-2.15.so
--snip--
vi    22728  juser cwd  DIR    8,1   4096 14945078 /home/juser/w/c
vi    22728  juser  4u  REG    8,1   1288  1056519 /home/juser/w/c/f
--snip--
```

The output shows the following fields (listed in the top row):

- **COMMAND**. The command name for the process that holds the file descriptor.
- **PID**. The process ID.
- **USER**. The user running the process.
- **FD**. This field can contain two kinds of elements. In the output above, the FD column shows the purpose of the file. The FD field can also list the *file descriptor* of the open file—a number that a process uses together with the system libraries and kernel to identify and manipulate a file.
- **TYPE**. The file type (regular file, directory, socket, and so on).
- **DEVICE**. The major and minor number of the device that holds the file.
- **SIZE**. The file's size.
- **NODE**. The file's inode number.
- **NAME**. The filename.

The lsof(1) manual page contains a full list of what you might see for each field, but you should be able to figure out what you're looking at just by looking at the output. For example, look at the entries with cwd in the FD field as highlighted in bold. These lines indicate the current working directories of the processes. Another example is the very last line, which shows a file that the user is currently editing with `vi`.

8.2.2 Using lsof

There are two basic approaches to running `lsof`:

- List everything and pipe the output to a command like `less`, and then search for what you're looking for. This can take a while due to the amount of output generated.
- Narrow down the list that `lsof` provides with command-line options.

You can use command-line options to provide a filename as an argument and have `lsof` list only the entries that match the argument. For example, the following command displays entries for open files in */usr*:

```
$ lsof /usr
```

To list the open files for a particular process ID, run:

```
$ lsof -p pid
```

For a brief summary of `lsof`'s many options, run `lsof -h`. Most options pertain to the output format. (See Chapter 10 for a discussion of the `lsof` network features.)

NOTE

`lsof` is highly dependent on kernel information. If you upgrade your kernel and you're not routinely updating everything, you might need to upgrade `lsof`. In addition, if you perform a distribution update to both the kernel and `lsof`, the updated `lsof` might not work until you reboot with the new kernel.

8.3 Tracing Program Execution and System Calls

The tools we've seen so far examine active processes. However, if you have no idea why a program dies almost immediately after starting up, even `lsof` won't help you. In fact, you'd have a difficult time even running `lsof` concurrently with a failed command.

The `strace` (system call trace) and `ltrace` (library trace) commands can help you discover what a program attempts to do. These tools produce extraordinarily large amounts of output, but once you know what to look for, you'll have more tools at your disposal for tracking down problems.

8.3.1 strace

Recall that a *system call* is a privileged operation that a user-space process asks the kernel to perform, such as opening and reading data from a file. The `strace` utility prints all the system calls that a process makes. To see it in action, run this command:

```
$ strace cat /dev/null
```

In Chapter 1, you learned that when one process wants to start another process, it invokes the `fork()` system call to spawn a copy of itself, and then the copy uses a member of the `exec()` family of system calls to start running a new program. The `strace` command begins working on the new process (the copy of the original process) just after the `fork()` call. Therefore, the first lines of the output from this command should show `execve()` in action, followed by a memory initialization call, `brk()`, as follows:

```
execve("/bin/cat", ["cat", "/dev/null"], [/* 58 vars */]) = 0

brk(0)                                  = 0x9b65000
```

The next part of the output deals primarily with loading shared libraries. You can ignore this unless you really want to know what the shared library system does.

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)

mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb77b5000

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)

open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

--snip--

open("/lib/libc.so.6", O_RDONLY)      = 3

read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200^\1"..., 1024)= 1024
```

In addition, skip past the `mmap` output until you get to the lines that look like this:

```
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0

open("/dev/null", O_RDONLY|O_LARGEFILE) = 3

fstat64(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0

fadvise64_64(3, 0, 0, POSIX_FADV_SEQUENTIAL)= 0

read(3,"", 32768)                        = 0

close(3)                                = 0

close(1)                                = 0

close(2)                                = 0

exit_group(0)                            = ?
```

This part of the output shows the command at work. First, look at the `open()` call, which opens a file. The `3` is a result that means success (`3` is the file descriptor that the kernel returns after opening the file). Below that, you see where `cat` reads from */dev/null* (the `read()` call, which also has `3` as the file descriptor). Then there's nothing more to read, so the program closes the file descriptor and exits with `exit_group()`.

What happens when there's a problem? Try `strace cat not_a_file` instead and examine the `open()` call in the resulting output:

```
open("not_a_file", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
```

Because `open()` couldn't open the file, it returned `-1` to signal an error. You can see that `strace` reports the exact error and gives you a small description of the error.

Missing files are the most common problems with Unix programs, so if the system log and other log information aren't very helpful and you have nowhere else to turn, `strace` can be of great use. You can even use it on daemons that detach themselves. For example:

```
$ strace -o crummyd_strace -ff crummyd
```

In this example, the `-o` option to `strace` logs the action of any child process that `crummyd` spawns into `crummyd_strace.pid`, where *`pid`* is the process ID of the child process.

8.3.2 ltrace

The `ltrace` command tracks shared library calls. The output is similar to that of `strace`, which is why we're mentioning it here, but it doesn't track anything at the kernel level. Be warned that there are *many* more shared library calls than system calls. You'll definitely need to filter the output, and `ltrace` itself has many built-in options to assist you.

NOTE

See 15.1.4 Shared Libraries for more on shared libraries. The `ltrace` command doesn't work on statically linked binaries.

8.4 Threads

In Linux, some processes are divided into pieces called *threads*. A thread is very similar to a process—it has an identifier (TID, or thread ID), and the kernel schedules and runs threads just like processes. However, unlike separate processes, which usually do not share system resources such as memory and I/O connections with other processes, all threads inside a single process share their system resources and some memory.

8.4.1 Single-Threaded and Multithreaded Processes

Many processes have only one thread. A process with one thread is *single-threaded*, and a process with more than one thread is *multithreaded*. All processes start out single-threaded. This starting thread is usually called the *main thread*. The main thread may then start new threads in order for the process to become multithreaded, similar to the way a process can call `fork()` to start a new process.

NOTE

It's rare to refer to threads at all when a process is single-threaded. This book will not mention threads unless multithreaded processes make a difference in what you see or experience.

The primary advantage of a multithreaded process is that when the process has a lot to do, threads can run simultaneously on multiple processors, potentially speeding up computation. Although you can also achieve simultaneous computation with multiple processes, threads start faster than processes, and it is often easier and/or more efficient for threads to intercommunicate using their shared memory than it is for processes to communicate over a channel such as a network connection or a pipe.

Some programs use threads to overcome problems managing multiple I/O resources. Traditionally, a process would sometimes use `fork()` to start a new subprocess in order to deal with a new input or output stream. Threads offer a similar mechanism without the overhead of starting a new process.

8.4.2 Viewing Threads

By default, the output from the `ps` and `top` commands shows only processes. To display the thread information in `ps`, add the `m` option. Here is some sample output:

Example 8-1. Viewing threads with ps m

```
$ ps m
  PID TTY      STAT   TIME COMMAND

 3587 pts/3    -      0:00 bash❶

    - -        Ss     0:00 -

 3592 pts/4    -      0:00 bash❷
```

```
    - -        Ss    0:00 -
12287 pts/8    -     0:54 /usr/bin/python /usr/bin/gm-notify❸
    - -        SLl   0:48 -
    - -        SLl   0:00 -
    - -        SLl   0:06 -
    - -        SLl   0:00 -
```

Example 8-1 shows processes along with threads. Each line with a number in the PID column (at ❶, ❷, and ❸) represents a process, as in the normal `ps` output. The lines with the dashes in the PID column represent the threads associated with the process. In this output, the processes at ❶ and ❷ have only one thread each, but process 12287 at ❸ is multithreaded with four threads.

If you would like to view the thread IDs with `ps`, you can use a custom output format. This example shows only the process IDs, thread IDs, and command:

Example 8-2. Showing process IDs and thread IDs with ps m

```
$ ps m -o pid,tid,command
  PID   TID   COMMAND
 3587     -   bash
    -  3587   -
 3592     -   bash
    -  3592   -
 12287    -   /usr/bin/python /usr/bin/gm-notify
    - 12287   -
    - 12288   -
    - 12289   -
    - 12295   -
```

The sample output in Example 8-2 corresponds to the threads shown in Example 8-1. Notice that the thread IDs of the single-threaded processes are identical to the process IDs; this is the main thread. For the multithreaded process 12287, thread 12287 is also the main thread.

NOTE

Normally, you won't interact with individual threads as you would processes. You need to know a lot about how a multithreaded program was written in order to act on one thread at a time, and even then, doing so might not be a good idea.

Threads can confuse things when it comes to resource monitoring because individual threads in a multithreaded process can consume resources simultaneously. For example, `top` doesn't show threads by default; you'll need to press H to turn it on. For most of the resource monitoring tools that you're about to see, you'll have to do a little extra work to turn on the thread display.

8.5 Introduction to Resource Monitoring

Now we'll discuss some topics in resource monitoring, including processor (CPU) time, memory, and disk I/O. We'll examine utilization on a systemwide scale, as well as on a per-process basis.

Many people touch the inner workings of the Linux kernel in the interest of improving performance. However, most Linux systems perform well under a distribution's default settings, and you can spend days trying to tune your machine's performance without meaningful results, especially if you don't know what to look for. So rather than think about performance as you experiment with the tools in this chapter, think about seeing the kernel in action as it divides resources among processes.

8.6 Measuring CPU Time

To monitor one or more specific processes over time, use the `-p` option to `top`, with this syntax:

```
$ top -p pid1 [-p pid2 ...]
```

To find out how much CPU time a command uses during its lifetime, use `time`. Most shells have a built-in `time` command that doesn't provide extensive statistics, so you'll probably need to run `/usr/bin/time`. For example, to measure the CPU time used by `ls`, run

```
$ /usr/bin/time ls
```

After `ls` terminates, `time` should print output like that below. The key fields are in boldface:

```
0.05user   0.09system   0:00.44elapsed   31%CPU   (0avgtext+0avgdata
0maxresident)k

0inputs+0outputs (125major+51minor)pagefaults 0swaps
```

- **User time**. The number of seconds that the CPU has spent running the program's *own* code. On modern processors, some commands run so quickly, and therefore the CPU time is so low, that `time` rounds down to zero.
- **System time**. How much time the kernel spends doing the process's work (for example, reading files and directories).
- **Elapsed time**. The total time it took to run the process from start to finish, including the time that the CPU spent doing other tasks. This number is normally not very useful for performance measurement, but subtracting the user and system time from elapsed time can give you a general idea of how long a process spends waiting for system resources.

The remainder of the output primarily details memory and I/O usage. You'll learn more about the page fault output in 8.9 Memory.

8.7 Adjusting Process Priorities

You can change the way the kernel schedules a process in order to give the process more or less CPU time than other processes. The kernel runs each process according to its scheduling *priority*, which is a number between –20 and 20, with –20 being the foremost priority. (Yes, this can be confusing.)

The `ps -l` command lists the current priority of a process, but it's a little easier to see the priorities in action with the `top` command, as shown here:

```
$ top

Tasks: 244 total,  2 running, 242 sleeping,   0 stopped,   0 zombie

Cpu(s): 31.7%us, 2.8%sy,  0.0%ni, 65.4%id,  0.2%wa,  0.0%hi,  0.0%si,
0.0%st

Mem:  6137216k total, 5583560k used,   553656k free,    72008k buffers

Swap:  4135932k total,  694192k used, 3441740k  free,   767640k cached

  PID  USER     PR NI  VIRT  RES  SHR S %CPU %MEM    TIME+ COMMAND

28883  bri       20  0 1280m 763m  32m S   58 12.7 213:00.65 chromium-
```

```
browse

 1175  root     20  0  210m  43m  28m R   44  0.7  14292:35 Xorg

 4022  bri       20  0  413m 201m  28m S   29  3.4   3640:13 chromium-
browse

 4029  bri       20  0  378m 206m  19m S    2  3.5  32:50.86 chromium-
browse

 3971  bri       20  0  881m 359m  32m S    2  6.0 563:06.88 chromium-
browse

 5378  bri      20  0  152m  10m 7064 S    1  0.2  24:30.21 compiz

 3821  bri      20  0  312m  37m  14m S    0  0.6  29:25.57 soffice.bin

 4117  bri       20  0  321m 105m  18m S    0  1.8  34:55.01 chromium-
browse

 4138  bri       20  0  331m  99m  21m S    0  1.7 121:44.19 chromium-
browse

 4274  bri       20  0  232m  60m  13m S    0  1.0  37:33.78 chromium-
browse

 4267  bri       20  0  1102m 844m 11m S    0 14.1  29:59.27 chromium-
browse

 2327  bri      20  0  301m  43m 16m S    0  0.7 109:55.65 unity-2d-shell
```

In the `top` output above, the PR (priority) column lists the kernel's current schedule priority for the process. The higher the number, the less likely the kernel is to schedule the process if others need CPU time. The schedule priority alone does not determine the kernel's decision to give CPU time to a process, and it changes frequently during program execution according to the amount of CPU time that the process consumes.

Next to the priority column is the *nice value* (NI) column, which gives a hint to the kernel's scheduler. This is what you care about when trying to influence the kernel's decision. The kernel adds the nice value to the current priority to determine the next time slot for the process.

By default, the nice value is 0. Now, say you're running a big computation in the background that you don't want to bog down your interactive session. To have that process take a backseat to other processes and run only when the other tasks have nothing to do, you could change the nice value to 20 with the `renice` command (where *`pid`* is the process ID of the process that you want to change):

```
$ renice 20 pid
```

If you're the superuser, you can set the nice value to a negative number, but doing so is almost always a bad idea because system processes may not get enough CPU time. In fact, you probably won't need to alter nice values much because many Linux systems have only a single user, and that user does not perform much real computation. (The nice value was much more important back when there were many users on a single machine.)

8.8 Load Averages

CPU performance is one of the easier metrics to measure. The *load average* is the average number of processes currently ready to run. That is, it is an estimate of the number of processes that are capable of using the CPU at any given time. When thinking about a load average, keep in mind that most processes on your system are usually waiting for input (from the keyboard, mouse, or network, for example), meaning that most processes are not ready to run and should contribute nothing to the load average. Only processes that are actually doing something affect the load average.

8.8.1 Using uptime

The `uptime` command tells you three load averages in addition to how long the kernel has been running:

```
$ uptime
... up 91 days, ... load average: 0.08, 0.03, 0.01
```

The three bolded numbers are the load averages for the past 1 minute, 5 minutes, and 15 minutes, respectively. As you can see, this system isn't very busy: An average of only 0.01 processes have been running across all processors for the past 15 minutes. In other words, if you had just one processor, it was only running user-space applications for 1 percent of the last 15 minutes. (Traditionally, most desktop systems would exhibit a load average of about 0 when you were doing anything *except* compiling a program or playing a game. A load average of 0 is usually a good sign, because it means that your processor isn't being challenged and you're saving power.)

NOTE

User interface components on current desktop systems tend to occupy more of the CPU than those in the past. For example, on Linux systems, a web browser's Flash plugin can be a particularly notorious resource hog, and Flash applications can easily occupy much of a system's CPU and memory due to poor all-around implementation.

If a load average goes up to around 1, a single process is probably using the CPU nearly all of the time. To identify that process, use the `top` command; the process will usually rise to the the top of the display.

Most modern systems have more than one processor core or CPU, so multiple processes can easily run simultaneously. If you have two cores, a load average of 1 means that only one of the cores is likely active at any given time, and a load average of 2 means that both cores have just enough to do all of the time.

8.8.2 High Loads

A high load average does not necessarily mean that your system is having trouble. A system with enough memory and I/O resources can easily handle many running processes. If your load average is high and your system still responds well, don't panic: The system just has a lot of processes sharing the CPU. The processes have to compete with each other for processor time, and as a result they'll take longer to perform their computations than they would if they were each allowed to use the CPU all of the time. Another case where you might see a high load average as normal is a web server, where processes can start and terminate so quickly that the load average measurement mechanism can't function effectively.

However, if you sense that the system is slow and the load average is high, you might be running into memory performance problems. When the system is low on memory, the kernel can start to *thrash*, or rapidly swap memory for processes to and from the disk. When this happens, many processes will become ready to run, but their memory might not be available, so they will remain in the ready-to-run state (and contribute to the load average) for much longer than they normally would.

We'll now look at memory in much more detail.

8.9 Memory

One of the simplest ways to check your system's memory status as a whole is to run the `free` command or view */proc/meminfo* to see how much real memory is being used for caches and buffers. As we've just mentioned, performance problems can arise from memory shortages. If there isn't much cache/buffer memory being used (and the rest of the real memory is taken), you may need more memory. However, it's too easy to blame a shortage of memory for every performance problem on your machine.

8.9.1 How Memory Works

Recall from Chapter 1 that the CPU has a memory management unit (MMU) that translates the virtual memory addresses used by processes into real ones. The kernel assists the MMU by breaking the memory used by processes into smaller chunks called *pages*. The kernel maintains a data structure, called a *page table*, that contains a mapping of a processes' virtual page addresses to real page addresses in memory. As a process accesses memory, the MMU translates the virtual addresses used by the process into real addresses based on the kernel's page table.

A user process does not actually need all of its pages to be immediately available in order to run. The kernel generally loads and allocates pages as a process needs them; this system is known as *on-demand paging* or just *demand paging*. To see how this works, consider how a program starts and runs as a new process:

1. The kernel loads the beginning of the program's instruction code into memory pages.
2. The kernel may allocate some working-memory pages to the new process.
3. As the process runs, it might reach a point where the next instruction in its code isn't in any of the pages that the kernel initially loaded. At this point, the kernel takes over, loads the necessary pages into memory, and then lets the program resume execution.
4. Similarly, if the program requires more working memory than was initially allocated, the kernel handles it by finding free pages (or by making room) and assigning them to the process.

8.9.2 Page Faults

If a memory page is not ready when a process wants to use it, the process triggers a *page fault*. In the event of a page fault, the kernel takes control of the CPU from the process in order to get the page ready. There are two kinds of page faults: minor and major.

Minor Page Faults

A minor page fault occurs when the desired page is actually in main memory but the MMU doesn't know where it is. This can happen when the process requests more memory or when the MMU doesn't have enough space to store all of the page locations for a process. In this case, the kernel tells the MMU about the page and permits the process to continue. Minor page faults aren't such a big deal, and many occur as a process runs. Unless you need maximum performance from some memory-intensive program, you probably shouldn't worry about them.

Major Page Faults

A major page fault occurs when the desired memory page isn't in main memory at all, which means that the kernel must load it from the disk or some other slow storage mechanism. A lot of major page faults will bog the system down because the kernel must do a substantial amount of work to provide the pages, robbing normal processes of their chance to run.

Some major page faults are unavoidable, such as those that occur when you load the code from disk when running a program for the first time. The biggest problems happen when you start running out of memory and the kernel starts to *swap* pages of working memory out to the disk in order to make room for new pages.

Watching Page Faults

You can drill down to the page faults for individual processes with the `ps`, `top`, and `time` commands. The following command shows a simple example of how the `time` command displays page faults. (The output of the cal command doesn’t matter, so we’re discarding it by redirecting that to */dev/null*.)

```
$ /usr/bin/time cal > /dev/null

0.00user    0.00system    0:00.06elapsed    0%CPU    (0avgtext+0avgdata
3328maxresident)k

648inputs+0outputs (2major+254minor)pagefaults 0swaps
```

As you can see from the bolded text, when this program ran, there were 2 major page faults and 254 minor ones. The major page faults occurred when the kernel needed to load the program from the disk for the first time. If you ran the command again, you probably wouldn’t get any major page faults because the kernel would have cached the pages from the disk.

If you’d rather see the page faults of processes as they’re running, use top or ps. When running top, use f to change the displayed fields and u to display the number of major page faults. (The results will show up in a new, nFLT column. You won’t see the minor page faults.)

When using ps, you can use a custom output format to view the page faults for a particular process. Here’s an example for process ID 20365:

```
$ ps -o pid,min_flt,maj_flt 20365

  PID  MINFL  MAJFL

20365 834182    23
```

The MINFL and MAJFL columns show the numbers of minor and major page faults. Of course, you can combine this with any other process selection options, as described in the ps(1) manual page.

Viewing page faults by process can help you zero in on certain problematic components. However, if you’re interested in your system performance as a whole, you need a tool to summarize CPU and memory action across all processes.

8.10 Monitoring CPU and Memory Performance with vmstat

Among the many tools available to monitor system performance, the vmstat command is one of the oldest, with minimal overhead. You’ll find it handy for getting a high-level view of how often the kernel is swapping pages in and out, how busy the CPU is, and IO utilization.

The trick to unlocking the power of vmstat is to understand its output. For example, here’s some output from vmstat 2, which reports statistics every 2 seconds:

```
$ vmstat 2

procs -----------memory---------- ---swap-- -----io---- -system-- ----
cpu----

 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa

 2  0 320416 3027696 198636 1072568    0    0     1     1    2    0 15  2 83  0

 2  0 320416 3027288 198636 1072564    0    0     0  1182  407  636  1  0 99  0

 1  0 320416 3026792 198640 1072572    0    0     0    58  281  537  1  0 99  0

 0  0 320416 3024932 198648 1074924    0    0     0   308  318  541  0  0 99  1

 0  0 320416 3024932 198648 1074968    0    0     0     0  208  416  0  0 99  0
 0  0 320416 3026800 198648 1072616   0   0    0    0 207 389  0  0 100 0
```

The output falls into categories: `procs` for processes, `memory` for memory usage, `swap` for the pages pulled in and out of swap, `io` for disk usage, `system` for the number of times the kernel switches into kernel code, and `cpu` for the time used by different parts of the system.

The preceding output is typical for a system that isn't doing much. You'll usually start looking at the second line of output—the first one is an average for the entire uptime of the system. For example, here the system has 320416KB of memory swapped out to the disk (`swpd`) and around 3025000KB (3 GB) of real memory `free`. Even though some swap space is in use, the zero-valued `si` (swap-in) and `so` (swap-out) columns report that the kernel is not currently swapping anything in or out from the disk. The `buff` column indicates the amount of memory that the kernel is using for disk buffers (see 4.2.5 Disk Buffering, Caching, and Filesystems).

On the far right, under the CPU heading, you see the distribution of CPU time in the `us`, `sy`, `id`, and `wa` columns. These list (in order) the percentage of time that the CPU is spending on user tasks, system (kernel) tasks, idle time, and waiting for I/O. In the preceding example, there aren't too many user processes running (they're using a maximum of 1 percent of the CPU); the kernel is doing practically nothing, while the CPU is sitting around doing nothing 99 percent of the time.

Now, watch what happens when a big program starts up sometime later (the first two lines occur right before the program runs):

Example 8-3. Memory activity

```
procs -----------memory---------- ---swap-- -----io---- -system-- ----
cpu----
r b    swpd   free   buff   cache  si   so   bi    bo   in   cs us sy id wa
1 0  320412  2861252  198920 1106804   0   0     0    0 2477 4481 25 2 72

0❶

1 0  320412  2861748  198924 1105624   0   0     0   40 2206 3966 26 2 72 0
1 0  320412  2860508  199320 1106504   0   0   210   18 2201 3904 26 2 71 1
1 1  320412  2817860  199332 1146052   0   0 19912    0 2446 4223 26 3 63 8
2 2  320284  2791608  200612 1157752 202   0  4960  854 3371 5714 27 3 51

18❷

1 1  320252  2772076  201076 1166656  10   0  2142 1190 4188 7537 30 3 53
14
0 3  320244  2727632  202104 1175420  20   0  1890  216 4631 8706 36 4 46
14
```

As you can see at ❶ in Example 8-3, the CPU starts to see some usage for an extended period, especially from user processes. Because there is enough free memory, the amount of cache and buffer space used starts to increase as the kernel starts to use the disk more.

Later on, we see something interesting: Notice at ❷ that the kernel pulls some pages into memory that were once swapped out (the `si` column). This means that the program that just ran probably accessed some pages shared by another process. This is common; many processes use the code in certain shared libraries only when starting up.

Also notice from the `b` column that a few processes are *blocked* (prevented from running) while waiting for memory pages. Overall, the amount of free memory is decreasing, but it's nowhere near being depleted. There's also a fair amount of disk activity, as seen by the increasing numbers in the `bi` (blocks in) and `bo` (blocks out) columns.

The output is quite different when you run out of memory. As the free space depletes, both the buffer and cache sizes decrease because the kernel increasingly needs the space for user processes. Once there is nothing left, you'll start to see activity in the `so` (swapped out) column as the kernel starts moving pages onto the disk, at which point nearly all of the other output columns change to reflect the amount of work that the kernel is doing. You see more system time, more data going in and out of the disk, and more processes blocked because the memory they want to use is not available (it has been swapped out).

We haven't explained all of the `vmstat` output columns. You can dig deeper into them in the `vmstat(8)` manual page, but you might have to learn more about kernel memory management first from a class or a book like *Operating System Concepts*, 9th edition (Wiley, 2012) in order to understand them.

8.11 I/O Monitoring

By default, `vmstat` shows you some general I/O statistics. Although you can get very detailed per-partition resource usage with `vmstat -d`, you'll get a lot of output from this option, which might be overwhelming. Instead, try starting out with a tool just for I/O called `iostat`.

8.11.1 Using iostat

Like `vmstat`, when run without any options, `iostat` shows the statistics for your machine's current uptime:

```
$ iostat
[kernel information]
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           4.46    0.01    0.67    0.31    0.00   94.55

Device:         tp s    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda            4.6 7      7.2 8       49.86   9493727   65011716
sde            0.0 0      0.0 0        0.00      1230          0
```

The `avg-cpu` part at the top reports the same CPU utilization information as other utilities that you've seen in this chapter, so skip down to the bottom, which shows you the following for each device:

tps	Average number of data transfers per second
kB_read/s	Average number of kilobytes read per second
kB_wrtn/s	Average number of kilobytes written per second
kB_read	Total number of kilobytes read
kB_wrtn	Total number of kilobytes written

Another similarity to `vmstat` is that you can give an interval argument, such as `iostat 2`, to give an update every 2 seconds. When using an interval, you might want to display only the device report by using the `-d` option (such as `iostat -d 2`).

By default, the `iostat` output omits partition information. To show all of the partition information, use the

-p ALL option. Because there are many partitions on a typical system, you'll get a lot of output. Here's part of what you might see:

```
$ iostat -p ALL

--snip

--Device:               tps       kB_read/s      kB_wrtn/s     kB_read
kB_wrtn

--snip-

sda                    4.67          7.27          49.83     9496139
65051472

sda1                   4.38          7.16          49.51     9352969
64635440

sda2                   0.00          0.00           0.00           6
0

sda5                   0.01          0.11           0.32      141884
416032

scd0                   0.00          0.00           0.00           0
0

--snip--

sde                    0.00          0.00           0.00        1230
0
```

In this example, sda1, sda2, and sda5 are all partitions of the sda disk, so there will be some overlap between the read and written columns. However, the sum of the partition columns won't necessarily add up to the disk column. Although a read from sda1 also counts as a read from sda, keep in mind that you can read from sda directly, such as when reading the partition table.

8.11.2 Per-Process I/O Utilization and Monitoring: iotop

If you need to dig even deeper to see I/O resources used by individual processes, the iotop tool can help. Using iotop is similar to using top. There is a continuously updating display that shows the processes using the most I/O, with a general summary at the top:

```
# iotop

Total DISK READ:       4.76 K/s | Total DISK WRITE:    333.31 K/s

  TID  PRIO  USER      DISK READ DISK  WRITE  SWAPIN     IO> COMMAND

  260 be/3 root         0.00 B/s  38.09 K/s  0.00 %  6.98 % [jbd2/sda1-
8]

 2611 be/4 juser         4.76 K/s  10.32 K/s  0.00 %  0.21 % zeitgeist-
daemon

 2636 be/4 juser         0.00 B/s  84.12 K/s  0.00 %  0.20 % zeitgeist-
fts

 1329 be/4 juser        0.00 B/s  65.87 K/s  0.00 %  0.03 % soffice.b~ash-
pipe=6
```

```
   6845 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % chromium-browser
  19069 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % rhythmbox
```

Along with the user, command, and read/write columns, notice that there is a `TID` column (thread ID) instead of a process ID. The `iotop` tool is one of the few utilities that displays threads instead of processes.

The `PRIO` (priority) column indicates the I/O priority. It's similar to the CPU priority that you've already seen, but it affects how quickly the kernel schedules I/O reads and writes for the process. In a priority such as `be/4`, the `be` part is the *scheduling class*, and the number is the priority level. As with CPU priorities, lower numbers are more important; for example, the kernel allows more time for I/O for a process with `be/3` than one with `be/4`.

The kernel uses the scheduling class to add more control for I/O scheduling. You'll see three scheduling classes from `iotop`:

- **`be`** Best-effort. The kernel does its best to fairly schedule I/O for this class. Most processes run under this I/O scheduling class.
- **`rt`** Real-time. The kernel schedules any real-time I/O before any other class of I/O, no matter what.
- **`idle`** Idle. The kernel performs I/O for this class only when there is no other I/O to be done. There is no priority level for the idle scheduling class.

You can check and change the I/O priority for a process with the `ionice` utility; see the ionice(1) manual page for details. You probably will never need to worry about the I/O priority, though.

8.12 Per-Process Monitoring with pidstat

You've seen how you can monitor specific processes with utilities such as `top` and `iotop`. However, this display refreshes over time, and each update erases the previous output. The `pidstat` utility allows you to see the resource consumption of a process over time in the style of `vmstat`. Here's a simple example for monitoring process 1329, updating every second:

```
$ pidstat -p 1329 1
Linux 3.2.0-44-generic-pae (duplex)     07/01/2015      _i686_ (4 CPU)
09:26:55 PM       PID    %usr %system %guest %CPU CPU Command
09:27:03 PM      1329    8.00  0.00     0.00 8.00   1 myprocess
09:27:04 PM      1329    0.00  0.00     0.00 0.00   3 myprocess
09:27:05 PM      1329    3.00  0.00     0.00 3.00   1 myprocess
09:27:06 PM      1329    8.00  0.00     0.00 8.00   3 myprocess
09:27:07 PM      1329    2.00  0.00     0.00 2.00   3 myprocess
09:27:08 PM      1329    6.00  0.00     0.00 6.00   2 myprocess
```

The default output shows the percentages of user and system time and the overall percentage of CPU time, and it even tells you which CPU the process was running on. (The `%guest` column here is somewhat odd—it's the percentage of time that the process spent running something inside a virtual machine. Unless you're running a virtual machine, don't worry about this.)

Although `pidstat` shows CPU utilization by default, it can do much more. For example, you can use the `-r` option to monitor memory and `-d` to turn on disk monitoring. Try them out, and then look at the pidstat(1) manual page to see even more options for threads, context switching, or just about anything else that we've