

Popular Unix Performance- Monitoring Tools for Linux

CHAPTER

3

Before the Linux kernel, there was Unix. Well, okay, there were a lot of different flavors of Unix, but most Unix systems came from Hewlett Packard's Unix HP-UX to Sun Microsystems's SunOS and Solaris. In addition to the commercial variety, there were also other free Unix operating systems. The Berkeley Software Distribution (BSD) was the main free Unix; it later spawned three flavors of free Unix systems: OpenBSD, FreeBSD, and NetBSD.

Without going into too much historical detail, it is safe to assume that the art of performance tuning has been around a long time. To better enable the system administrator in the task of performance tuning, many tools were created on these systems well before the Linux kernel became as popular as it is today. Almost all of those tools have been ported to or rewritten for use on Linux-based operating systems. The reverse also is true: Many of the newer tools initially developed on Linux-based systems have been ported on to other flavors of Unix. This chapter looks at some of the more popular Unix tools that have been ported or rewritten for use on Linux systems.

NOTE

For more information on the history of Unix, go online and search <http://www.yahoo.com/> or <http://www.google.com/>. Many Web sites chronicle Unix and Unix-like operating systems history.

The Scope of These Tools and the Chapter

Take my word for it: A plethora of other performance-monitoring tools are available for Linux and other Unix operating systems. This chapter takes a look at the most common of these tools. Indeed, an entire book could be written on every single performance tool ported to or written on Linux systems. The unfortunate truth, however, is that the book would never be up-to-date because so many are added almost daily.

This chapter is divided into four generic sections of tools:

1. All-purpose tools
2. Disk benchmark tools
3. Network monitoring tools
4. Other tools

It is important to note that we do not break down the tools by subsystem alone because there are very few subsystem-specific tools. Many performance-monitoring tools can monitor all the subsystems or, as was noted in Chapter 1, “Overview of Performance Tuning,” the big three: CPU, disk, and memory.

Interpreting Results and Other Notes

Before delving into how each tool works and what the returned information means, I have to throw out a disclaimer. Most performance-monitoring tools used in a somewhat random pattern can sometimes yield random results. It is quite common for documentation that comes with a tool to state that scripting with it might yield better long-term results.

To better understand this, think about real performance and perceived performance. Even the system administrator can be tricked into “freaking out” over a quick loss of performance. A prime example is with quick compiling programs. If you happen to run a monitoring tool around the same time that a programmer is running a compile, it might appear that the system is being taxed when, in fact, it is not under a sustained load. Most likely you already realize that systems will occasionally face a quick burst load that really does not harm the overall performance of the system. However, do all users or staff members realize this? Most likely they do not—remember, a little user education never hurts.

All-Purpose Tools

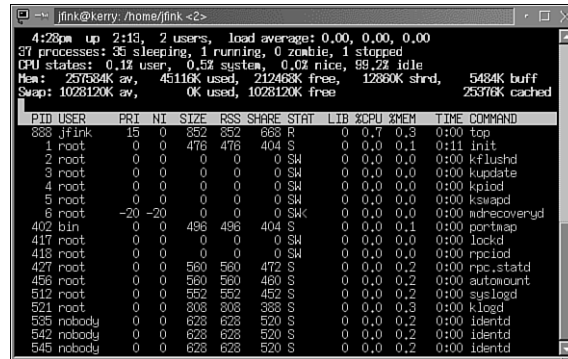
Unix provides many tools for the system administrator to monitor systems. However, it is worth noting at this point that what you read here about these utilities might be different, depending on your Linux distribution.

The top Utility

Arguably one of the most popular tools for performance monitoring on Unix systems in general is the top utility. As its name might suggest, this tool displays the top CPU-intensive processes in close to real time. The top display is refreshed every 5 seconds by default, but it can be modified with the `-s` option for longer or shorter intervals. The top utility also provides an interactive mode for modifying process behavior. The following is the syntax for starting top:

```
$ top
```

Figure 3.1 shows an example of top output to a standard terminal screen.

**FIGURE 3.1**

An example of the top process monitor.

Table 3.1 explains the monitor fields in top.

TABLE 3.1 top Monitor Fields

Field	Description
up	How long the system has been running since the last time it was rebooted. The load averages are displayed as well.
processes	Total number of processes running since the last time the top monitor checked. This measurement includes all processes on the system.
CPU States	An average percent of CPU time. This field examines all aspects of CPU time, including user, idle, and niced tasks. Because niced tasks are included, the total can go over 100%, so do not be alarmed if this is the case (unless it is an outrageous value, such as 160%).
Mem	All the memory statistics, such as total memory, available memory for nonkernel processes, memory in use, and memory that is shared and buffered.
Swap	Swap statistics, including total allocated swap space, available swap space, and used swap space.

Table 3.2 explains the process fields in top.

TABLE 3.2 top Process Fields

Field	Description
PID	Process ID of each task.
USER	Username of each task.

TABLE 3.2 Continued

<i>Field</i>	<i>Description</i>
PRI	Priority of each task.
NI	Nice value of each task.
SIZE	Total size of a task, including code and data, plus the stack space in kilobytes.
RSS	Amount of physical memory used by the task.
SHARE	Amount of shared memory used by a task.
STATE	Current CPU state of a task. The states can be S for sleeping, D for uninterruptible, R for running, T for stopped/traced, and Z for zombied.
TIME	The CPU time that a task has used since it started.
%CPU	The CPU time that a task has used since the last update.
%MEM	A task's share of physical memory.
COMMAND	The task's command name.

NOTE

A task's command name is truncated if the tasks have only the program name in parentheses.

In addition to just watching the top display, you can manipulate the top display in interactive mode and modify running processes. The interactive mode is invoked by pressing H while the display is running.

What about understanding what top tells you and knowing how to filter? A good example is simple filtering. In this example, a process is intermittently jumping to the top of the monitor. It might help to stop the monitor and start it with a few filtering options to get rid of information that you do not want:

```
$ top -i
```

Still, you cannot catch the culprit in action. Next you can disable some of the information that you want by toggling on or off displays such as the memory summary.

The great thing about top is that it provides a lot of different information quickly, and it updates periodically. For example, a process might be at the top, but why? Is it because it requires more processor time? Or is it eating memory? With the additional fields shown, more information is displayed in one location.

vmstat

The name `vmstat` comes from “report virtual memory statistics.” The `vmstat` utility does a bit more than this, though. In addition to reporting virtual memory, `vmstat` reports certain kernel statistics about processes, disk, trap, and CPU activity.

The syntax for `vmstat` is as follows:

```
$ vmstat interval [count]
```

A sample syntax with an interval of 5 seconds and five counts would look like this:

```
$ vmstat 5 5
procs
r  b  w    swpd   free   buff  cache   si   so    bi    bo    in    cs   us   sy   id
0  0  0     380    3760  71616  944176   0   0    24   13   40   45   0   0   32
0  0  0     380    3760  71616  944176   0   0     0    4   105   5   0   1   99
0  0  0     380    3760  71616  944176   0   0     0    1  105   4   0   0  100
0  0  0     380    3760  71616  944176   0   0     0    0  103   4   0   0  100
0  0  0     380    3760  71616  944176   0   0     0    0  103   5   0   0  100
```

The very first line of output by `vmstat` is the average values for statistics since boot time, so do not be alarmed by high values.

The `vmstat` output is actually broken up into five sections: `procs`, `memory`, `swap`, `io`, and `cpu`.

Each section is outlined in the following table.

TABLE 3.3 The `procs` Section

<i>Field</i>	<i>Description</i>
<code>r</code>	Number of processes that are in a wait state and basically not doing anything but waiting to run
<code>b</code>	Number of processes that were in sleep mode and were interrupted since the last update
<code>w</code>	Number of processes that have been swapped out by <code>mm</code> and <code>vm</code> subsystems and have yet to run

TABLE 3.4 The `Memory` Section

<i>Field</i>	<i>Description</i>
<code>swpd</code>	The total amount of physical virtual memory in use
<code>free</code>	The amount of physical memory that is free or available
<code>buff</code>	Memory that was being buffered when the measurement was taken
<code>cache</code>	Cache that is in use

TABLE 3.5 The Swap Section

<i>Field</i>	<i>Description</i>
si	Amount of memory transferred from swap space back into memory
so	Amount of memory swapped to disk

TABLE 3.6 The IO Section

<i>Field</i>	<i>Description</i>
bi	Disk blocks sent to disk devices in blocks per second

TABLE 3.7 The System Section

<i>Field</i>	<i>Description</i>
in	Interrupts per second, including the CPU clocks
cs	Context switches per second within the kernel

TABLE 3.8 The CPU Section

<i>Field</i>	<i>Description</i>
us	Percentage of CPU cycles spent on user processes
sy	Percentage of CPU cycles spent on system processes
id	Percentage of unused CPU cycles or idle time when the CPU is basically doing nothing

Interpreting the output of these measurements tells you that there is some outbound disk activity (as shown by the bo field). Also note the increase in the in field of the system section once vmstat was started. This implies that vmstat increases the number of interrupts. Here you return to the idea of properly reading output. It is a good idea to look at the tools covered in this chapter when the system is idling so that you know what effect a performance-monitoring tool can have on the system.

xload and xosview

Along with many command-line-driven tools for Unix systems are X11-based tools. This section covers two of them, xload and xosview. Both distinctly different graphical monitoring tools, they provide a very quick glance method of watching your systems run.

xload

The first of the two is xload. In a nutshell, xload displays the system load average. It also has an alarm bar that shows when the load average is high. Figure 3.2 shows what xload typically looks like.

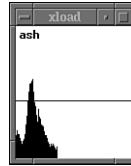


FIGURE 3.2

An example of xload.

To start xload, simply open an xterminal on the system and type the following:

```
$ xload &
```

The system knows to which display to go, so it automatically pops up. If you are running xload from a remote system to your X Window System, you can do one of two things:

1. Set a permanent DISPLAY variable in your .profile by entering this:

```
DISPLAY=<your_ip_address>:0  
export DISPLAY
```

2. Simply direct it to your system with this syntax:

```
$ xload -display <your_ip_address> &
```

The xload utility displays a histogram (chart) of the system load average. It updates this information periodically. The default is every 10 seconds, but this can be changed with the `-update` option.

xosview

The `xosview` utility can display a much more detailed collection of histograms about the system's performance in close to real time.

Of course, you easily can filter what you want to see in `xosview` using command-line options by specifying the subsystem with `+` or `-` (true and false, respectively). For example, to not see the CPU histogram, the syntax would be this:

```
$ xosview -cpu &
```

The `xosview` utility can be started from remote systems to a local X server in the same manner as `xload`. The colors, font, and other properties can be controlled by Xresource settings.

**FIGURE 3.3**

xosview, with all subsystems being monitored.

One interesting aspect of *xosview* is that it combines certain histograms into one horizontal bar. This requires some explanation.

- **LOAD**—This field has two colors. The first color is the load average of processes, and the second color (the background color) is idle processes relative to the load average. When the load average goes above 1.0, the bar will change colors.
- **CPU**—The CPU field has four colors related to process type usage: *usr*, *nice*, *sys*, and *free*.
- **MEM**—In this field, the full amount of real memory in use is shown. There are four colors in this field: user-allocated memory, shared memory, buffers, and free memory.
- **SWAP**—Swap has two colors; the first indicates the swap in use, and the second indicates what is free.
- **PAGE**—This field has three colors: *in*, for paging in; *out*, for paging out; and *idle*.
- **DISK**—This field has three colors: *in*, for transfers to disk; *out*, for transfers from disk; and *idle*.
- **INTS**—This field is a set of indicators that correspond to IRQ usage. They are numbered starting at 0 from the left.

uptime

The *uptime* command displays the current time, the length of time that the system has been up, the number of users, and the load average of the system over the last 1, 5, and 15 minutes. It looks something like this:

```
6:51PM up 2 days, 22:50, 6 users, load averages: 0.18, 0.30, 0.34
```

Benchmarking Your Disks with Bonnie

Most disk performance-monitoring tools are built into other performance-monitoring tools (take *vmstat*, for example). However, there is another approach to take. Benchmarking tools can help ascertain a performance bottleneck, even though the one that will be discussed in this

section actually can cause somewhat of a bottleneck itself. At the least, it is a good way to test systems before they are put into production.

The particular tool that this section addresses is called *bonnie*. The *bonnie* utility runs a performance test of filesystem I/O; it uses standard C library calls. *Bonnie* writes 8KB blocks in an attempt to discover the maximum sustained rate of transfer. As an added bonus, it cycles through rereading and rewriting to accurately simulate filesystem usage.

To use *bonnie*, the syntax is pretty simple:

```
bonnie -d <scratch_directory> -s <size_in_MB_of_testfiles> -m
↳<machine_label>
```

If no directory is specified, *bonnie* writes to the current working directory. If no size is given, 10MB is used. The machine label generally does not matter.

The following is some sample output of *bonnie* on a Sun SparcSTATION5:

```
$ bonnie
File './Bonnie.2831', size: 104857600
Writing with putc()...done
Rewriting...done
Writing intelligently...done
Reading with getc()...done
Reading intelligently...done
Seeker 1...Seeker 2...Seeker 3...start 'em...done...done...done...
-----Sequential Output----- ---Sequential Input-- --Random--
-Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --Seeks---
Machine    MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU /sec %CPU
          100  1650 65.0  1791 12.2  1141 14.1  2379 88.3  3285 20.4  62.5  4.9
$
```

As you can see, the fields of the final output are very self-explanatory. Now here are the results of *bonnie* on an x86 platform:

```
$ bonnie
File './Bonnie.22239', size: 104857600
Writing with putc()...done
Rewriting...done
Writing intelligently...done
Reading with getc()...done
Reading intelligently...done
Seeker 1...Seeker 2...Seeker 3...start 'em...done...done...done...
-----Sequential Output----- ---Sequential Input-- --Random--
-Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --Seeks---
Machine    MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU /sec %CPU
          100  2204 12.5  2244  4.3   925  3.1  2356 11.4  2375  6.3  43.0  1.1
$
```

Just a little different?

Tools like *bonnie* really help illustrate the difference between an untuned system and a tuned one, or an older (in this case, the *sparc* system) and a new one (and *x86* box).

Other Tools

In addition to the general tools and subsystem-specific ones, you have a variety of mixed and other performance-monitoring tools at your disposal. The next sections look at these in more detail.

ps

The *ps* command is another highly used tool where performance is concerned. Most often it is used to isolate a particular process. However, it also has numerous options that can help you get more out of *ps* and perhaps save some time while trying to isolate a particular process.

The *ps* command basically reports process status. When invoked without any options, the output looks something like this:

```
$ ps
  PID TTY          TIME CMD
 3220 pts/0    00:00:00 bash
 3251 pts/0    00:00:00 ps
```

This basically tells you everything that the current session of the user who invoked it is doing.

Obviously, just seeing what you are doing in your current session is not always all that helpful—unless, of course, you are doing something very detrimental in the background!

To look at other users or the system as a whole, *ps* requires some further options. The *ps* command's options on Linux are actually grouped into sections based on selection criteria.

Let's look at these sections and what they can do.

Simple Process Selection

Using simple process selection, you can be a little selective about what you see. For example, if you want to see only processes that are attached to your current terminal, you would use the *-T* option:

```
[jfink@kerry jfink]$ ps -T
  PID TTY          STAT TIME  COMMAND
 1668 pts/0        S      0:00  login -- jfink
 1669 pts/0        S      0:00  -bash
 1708 pts/0        R      0:00  ps -T
```

Process Selection by List

Another way to control what you see with `ps` is to view by a list type. As an example, if you want to see all the `identd` processes running, you would use the `-C` option from this group that displays a given command:

```
[jfink@kerry jfink]$ ps -C identd
  PID TTY          TIME CMD
  535 ?            00:00:00 identd
  542 ?            00:00:00 identd
  545 ?            00:00:00 identd
  546 ?            00:00:00 identd
  550 ?            00:00:00 identd
```

Output Format Control

Following process selection is output control. This is helpful when you want to see information in a particular format. A good example is using the jobs format with the `-j` option:

```
[jfink@kerry jfink]$ ps -j
  PID  PGID   SID  TTY          TIME CMD
 1669  1669  1668 pts/0      00:00:00 bash
 1729  1729  1668 pts/0      00:00:00 ps
```

Output Modifiers

Output modifiers can apply high-level changes to the output. The following is the output using the `-e` option to show the environment after running `ps`:

```
[jfink@kerry jfink]$ ps ae
  PID TTY      STAT   TIME COMMAND
 1668 pts/0    S       0:00 login -- jfink
 1669 pts/0    S       0:00 -bash TERM=ansi REMOTEHOST=172.16.14.102
HOME=/home/j
 1754 pts/0    R       0:00 ps ae LESSOPEN=|/usr/bin/lesspipe.sh %s
```

The remaining sections are `INFORMATION`, which provides versioning information and help, and `OBSOLETE` options. The next three sections give some specific cases of using `ps` with certain options.

Some Sample `ps` Output

Of course, reading the man page helps, but a few practical applied examples always light the way a little better.

The most commonly used `ps` switch on Linux and BSD systems is this:

```
$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  1116   380 ?        S    Jan27   0:01 init [3]
```

```

root      2  0.0  0.0    0    0 ?      SW   Jan27   0:03 [kflushd]
root      3  0.0  0.0    0    0 ?      SW   Jan27   0:18 [kupdate]
root      4  0.0  0.0    0    0 ?      SW   Jan27   0:00 [kpiod]
root      5  0.0  0.0    0    0 ?      SW   Jan27   0:38 [kswapd]
bin     260  0.0  0.0  1112  452 ?      S    Jan27   0:00 portmap
root    283  0.0  0.0  1292  564 ?      S    Jan27   0:00 syslogd -m 0
root    294  0.0  0.0  1480  700 ?      S    Jan27   0:00 klogd
daemon  308  0.0  0.0  1132  460 ?      S    Jan27   0:00 /usr/sbin/atd
root    322  0.0  0.0  1316  460 ?      S    Jan27   0:00 crond
root    322  0.0  0.0  1316  460 ?      S    Jan27   0:00 crond
root    336  0.0  0.0  1260  412 ?      S    Jan27   0:00 inetd
root    371  0.0  0.0  1096  408 ?      S    Jan27   0:00 rpc.rquotad
root    382  0.0  0.0  1464  160 ?      S    Jan27   0:00 [rpc.mountd]
root    393  0.0  0.0    0    0 ?      SW   Jan27   2:15 [nfsd]
root    394  0.0  0.0    0    0 ?      SW   Jan27   2:13 [nfsd]
root    395  0.0  0.0    0    0 ?      SW   Jan27   2:13 [nfsd]
root    396  0.0  0.0    0    0 ?      SW   Jan27   2:12 [nfsd]
root    397  0.0  0.0    0    0 ?      SW   Jan27   2:12 [nfsd]
root    398  0.0  0.0    0    0 ?      SW   Jan27   2:12 [nfsd]
root    399  0.0  0.0    0    0 ?      SW   Jan27   2:11 [nfsd]
root    400  0.0  0.0    0    0 ?      SW   Jan27   2:14 [nfsd]
root    428  0.0  0.0  1144  488 ?      S    Jan27   0:00 gpm -t ps/2
root    466  0.0  0.0  1080  408 tty1    S    Jan27   0:00 /sbin/mingetty
└─tt
root    467  0.0  0.0  1080  408 tty2    S    Jan27   0:00 /sbin/mingetty
└─tt
root    468  0.0  0.0  1080  408 tty3    S    Jan27   0:00 /sbin/mingetty
└─tt
root    469  0.0  0.0  1080  408 tty4    S    Jan27   0:00 /sbin/mingetty
└─tt
root    470  0.0  0.0  1080  408 tty5    S    Jan27   0:00 /sbin/mingetty
└─tt
root    471  0.0  0.0  1080  408 tty6    S    Jan27   0:00 /sbin/mingetty
└─tt
root   3326  0.0  0.0  1708  892 ?      R    Jan30   0:00 in.telnetd
root   3327  0.0  0.1  2196 1096 pts/0    S    Jan30   0:00 login -- jfink
jfink   3328  0.0  0.0  1764 1012 pts/0    S    Jan30   0:00 -bash
jfink   3372  0.0  0.0  2692 1008 pts/0    R    Jan30   0:00 ps aux

```

The output implies that this system's main job is to serve files via NFS, and indeed it is. It also doubles as an FTP server, but no connections were active when this output was captured.

The output of `ps` can tell you a lot more—sometimes just simple things that can improve performance. Looking at this NFS server again, you can see that it is not too busy; actually, it gets used only a few times a day. So what are some simple things that could be done to make it run even faster? Well, for starters, you could reduce the number of virtual consoles that are accessible via the system console. I like to have a minimum of three running (in case I lock one or

two). A total of six are shown in the output (the `mingetty` processes). There are also nine available `nfsd` processes; if the system is not used very often and only by a few users, that number can be reduced to something a little more reasonable.

Now you can see where tuning can be applied outside the kernel. Sometimes just entire processes do not need to be running, but those that require multiple instances (such as NFS, MySQL, or HTTP, for example) can be minimized to what is required for good operations.

The Process Forest

The process forest is a great way of seeing exactly how processes and their parents are related. The following output is a portion of the same system used in the previous section:

```
...
root      336  0.0  0.0 1260  412 ?          S   Jan27   0:00 inetd
root      3326 0.0  0.0 1708  892 ?          S   Jan30   0:00 \_ in.telnetd
root      3327 0.0  0.1 2196 1096 pts/0    S   Jan30   0:00      \_ login --
jfink     3328 0.0  0.0 1768 1016 pts/0    S   Jan30   0:00          \_ -
bash
jfink     3384 0.0  0.0 2680  976 pts/0    R   Jan30   0:00          \_
p
s
...
```

Based on that output, you easily can see how the system call `fork` got its name.

The application here is great. Sometimes a process itself is not to blame—and what if you kill an offending process only to find it respawned? The tree view can help track down the original process and kill it.

Singling Out a User

Last but definitely not least, you might need (or want) to look at a particular user's activities. On this particular system, my user account is the only `userland` account that does anything. I have chosen `root` to be the user to look at:

```
$ ps u --User root
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START    TIME COMMAND
root         1  0.0  0.0   1116   380 ?        S    Jan27    0:01 init [3]
root         2  0.0  0.0      0      0 ?        SW   Jan27    0:03 [kflushd]
root         3  0.0  0.0      0      0 ?        SW   Jan27    0:18 [kupdate]
root         4  0.0  0.0      0      0 ?        SW   Jan27    0:00 [kpiod]
root         5  0.0  0.0      0      0 ?        SW   Jan27    0:38 [kswapd]
root       283  0.0  0.0   1292   564 ?        S    Jan27    0:00 syslogd -m 0
root       294  0.0  0.0   1480   700 ?        S    Jan27    0:00 klogd
daemon    308  0.0  0.0   1132   460 ?        S    Jan27    0:00 /usr/sbin/atd
root      322  0.0  0.0   1316   460 ?        S    Jan27    0:00 crond
root      336  0.0  0.0   1260   412 ?        S    Jan27    0:00 inetd
```

```

root      350  0.0  0.0  1312  512 ?          S   Jan27   0:00 lpd
root      371  0.0  0.0  1096  408 ?          S   Jan27   0:00 rpc.rquotad
root      382  0.0  0.0  1464  160 ?          S   Jan27   0:00 [rpc.mountd]
root      393  0.0  0.0      0   0 ?          SW  Jan27   2:15 [nfsd]
root      394  0.0  0.0      0   0 ?          SW  Jan27   2:13 [nfsd]
root      395  0.0  0.0      0   0 ?          SW  Jan27   2:13 [nfsd]
root      396  0.0  0.0      0   0 ?          SW  Jan27   2:12 [nfsd]
root      397  0.0  0.0      0   0 ?          SW  Jan27   2:12 [nfsd]
root      398  0.0  0.0      0   0 ?          SW  Jan27   2:12 [nfsd]
root      399  0.0  0.0      0   0 ?          SW  Jan27   2:11 [nfsd]
root      400  0.0  0.0      0   0 ?          SW  Jan27   2:14 [nfsd]
root      428  0.0  0.0  1144  488 ?          S   Jan27   0:00 gpm -t ps/2
root      466  0.0  0.0  1080  408 tty1        S   Jan27   0:00 /sbin/mingetty
tt
y
root      467  0.0  0.0  1080  408 tty2        S   Jan27   0:00 /sbin/mingetty
tt
y
root      468  0.0  0.0  1080  408 tty3        S   Jan27   0:00 /sbin/mingetty
tt
y
root      469  0.0  0.0  1080  408 tty4        S   Jan27   0:00 /sbin/mingetty
tt
y
root      470  0.0  0.0  1080  408 tty5        S   Jan27   0:00 /sbin/mingetty
tt
y
root      471  0.0  0.0  1080  408 tty6        S   Jan27   0:00 /sbin/mingetty
tt
y
root      3326 0.0  0.0  1708  892 ?          R   Jan30   0:00 in.telnetd
root      3327 0.0  0.1  2196 1096 pts/0        S   Jan30   0:00 login - jfink

```

Applying only a single user's process is helpful when a user might have a runaway. Here's a quick example: A particular piece of software used by the company for which I work did not properly die when an attached terminal disappeared (it has been cleaned up since then). It collected error messages into memory until it was killed. To make matters worse, these error message went into shared memory queues.

The only solution was for the system administrator to log in and kill the offending process. Of course, after a period of time, a script was written that would allow users to do this in a safe manner. On this particular system, there were thousands of concurrent processes. Only by filtering based on the user or doing a grep from the whole process table was it possible to figure out which process it was and any other processes that it might be affecting.

free

The `free` command rapidly snags information about the state of memory on your Linux system. The syntax for `free` is pretty straightforward:

```
$ free
```

The following is an example of `free`'s output:

```
$ free
              total        used        free      shared    buffers     cached
Mem:      1036152      1033560         2592        8596       84848       932080
-/+ buffers/cache:      16632      1019520
Swap:      265064         380       264684
```

The first line of output shows the physical memory, and the last line shows similar information about swap. Table 3.9 explains the output of `free`.

TABLE 3.9 `free` Command Output Fields

<i>Field</i>	<i>Description</i>
total	Total amount of user available memory, excluding the kernel memory. (Don't be alarmed when this is lower than the memory on the machine.)
used	Total amount of used memory.
free	Total amount of memory that is free.
shared	Total amount of shared memory that is in use.
buffers	Current size of the disk buffer cache.
cached	Amount of memory that has been cached off onto disk.

An analysis of the sample output shows that this system seems to be pretty healthy. Of course, this is only one measurement. What if you want to watch the memory usage over time? The `free` command provides an option to do just that: the `-s` option. The `-s` option activates polling at a specified interval. The following is an example:

```
[jfink@kerry jfink]$ free -s 60
total        used        free      shared    buffers     cached
Mem:      257584      65244      192340        12132       40064       4576
-/+ buffers/cache:      20604      236980
Swap:      1028120         0      1028120

              total        used        free      shared    buffers     cached
Mem:      257584      66424      191160        12200       40084       5728
-/+ buffers/cache:      20612      236972
```



```

Swap:      1028120          0      1028120

           total        used        free      shared    buffers     cached
Mem:       257584        66528       191056       12200        40084        5812
-/+ buffers/cache:      20632       236952
Swap:      1028120          0      1028120
...

```

To stop `free` from polling, hit an interrupt key.

These measurements show a pretty quiet system, but the `free` command can come in handy if you want to see the effect of one particular command on the system. Run the command when the system is idling, and poll memory with `free`. `free` is well suited for this because of the granularity that you get in the output.

time

One very simple tool for examining the system is the `time` command. The `time` command comes in handy for relatively quick checks of how the system performs when a certain command is invoked. The way this works is simple: `time` returns a string value with information about the process and is launched with process like this:

```
$ time <command_name> [options]
```

Here is an example:

```
$ time cc hello.c -o hello
```

The output from the `time` command looks like this:

```

$ time cc hello.c -o hello
0.08user 0.04system 0:00.11elapsed 107%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (985major+522minor)pagefaults 0swaps

```

Even though this output is quite low-level, the `time` command can return very enlightening information about a particular command or program. It becomes very helpful in large environments in which operations normally take a long time. An example of this is comparing kernel compile times between different machines.

Some Network-Monitoring Tools

Many times system performance can be relative to external factors such as the network. Unix has a vast array of tools to examine network performance, from single host-monitoring software to applications that can monitor and manage vast WANs. This section looks at four relatively low-key applications for monitoring network activity:

- ping
- traceroute
- tcpdump
- ntop

ping

The ping utility is a very simple program that is most often used to simply see if a host is alive on the network. However, the return information from ping can often tell you how well a particular host-to-host connection is performing. The following is a sample of a ping session:

```
$ ping tesla
PING tesla.dp.asi (192.187.226.6): 56 data bytes
64 bytes from 192.187.226.6: icmp_seq=0 ttl=255 time=6.119 ms
64 bytes from 192.187.226.6: icmp_seq=1 ttl=255 time=0.620 ms
64 bytes from 192.187.226.6: icmp_seq=2 ttl=255 time=3.483 ms
64 bytes from 192.187.226.6: icmp_seq=3 ttl=255 time=1.340 ms
64 bytes from 192.187.226.6: icmp_seq=4 ttl=255 time=0.633 ms
64 bytes from 192.187.226.6: icmp_seq=5 ttl=255 time=7.803 ms
64 bytes from 192.187.226.6: icmp_seq=6 ttl=255 time=5.475 ms
--- tesla.dp.asi ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max/std-dev = 0.620/3.639/7.803/2.681 ms
```

Not too bad at all. Now I have purposely saturated the interface on host tesla (with several other pings running with the `-f` option to flood it at once, of course). Look at the results:

```
$ ping tesla
PING tesla.dp.asi (192.187.226.6): 56 data bytes
64 bytes from 192.187.226.6: icmp_seq=0 ttl=255 time=3.805 ms
64 bytes from 192.187.226.6: icmp_seq=1 ttl=255 time=1.804 ms
64 bytes from 192.187.226.6: icmp_seq=2 ttl=255 time=8.672 ms
64 bytes from 192.187.226.6: icmp_seq=3 ttl=255 time=1.616 ms
64 bytes from 192.187.226.6: icmp_seq=4 ttl=255 time=6.793 ms
64 bytes from 192.187.226.6: icmp_seq=5 ttl=255 time=1.607 ms
64 bytes from 192.187.226.6: icmp_seq=6 ttl=255 time=2.393 ms
64 bytes from 192.187.226.6: icmp_seq=7 ttl=255 time=1.601 ms
64 bytes from 192.187.226.6: icmp_seq=8 ttl=255 time=6.073 ms
64 bytes from 192.187.226.6: icmp_seq=9 ttl=255 time=1.615 ms
64 bytes from 192.187.226.6: icmp_seq=10 ttl=255 time=9.402 ms
64 bytes from 192.187.226.6: icmp_seq=11 ttl=255 time=1.875 ms
64 bytes from 192.187.226.6: icmp_seq=12 ttl=255 time=1.815 ms
--- tesla.dp.asi ping statistics ---
13 packets transmitted, 13 packets received, 0% packet loss
round-trip min/avg/max/std-dev = 0.601/2.774/8.402/2.802 ms
```

As you can see, there is a slightly higher time lapse, on average.
ping has a number of useful options. Table 3.10 lists some of them with example usage ideas:

TABLE 3.10 ping Options

-c	This option means that ping will ping a node only <i>c</i> times. This is especially useful for noninteractive scripts or if you are not interested in watching ping for more than a quick test to determine whether it is alive.
-f	The <i>flood</i> option sends requests as fast as the host you are pinging from can. This is a good way to measure just how well a host can send them. As mentioned in a previous example, it's also an easy way to load down a system for testing.
-r	This option bypasses routing tables and helps if a new node is on the network but the system you are pinging from either is not aware of it or is not in the same subnet.
-s	This option can modify the packet size and is useful for seeing whether the node that is being pinged is having issues with the size of packets being sent to it from other nodes.

traceroute

The traceroute utility is invaluable for discovering where a problem on a network might be located. It also is of great use in ensuring that your host is talking to the network just fine and that any problems might lie elsewhere. On this particular network, I show two traceroutes, one to a host that is local to the network and one to a remote host on a network in another city. The difference between these is astounding.

Here's the local traceroute:

```
$ traceroute andy
traceroute to strider.diverge.org (192.168.1.1), 64 hops max, 40 byte packets
1 strider.diverge.org (192.168.1.1) 0.547 ms 0.469 ms 3.383 ms
$
```

And here's the remote traceroute:

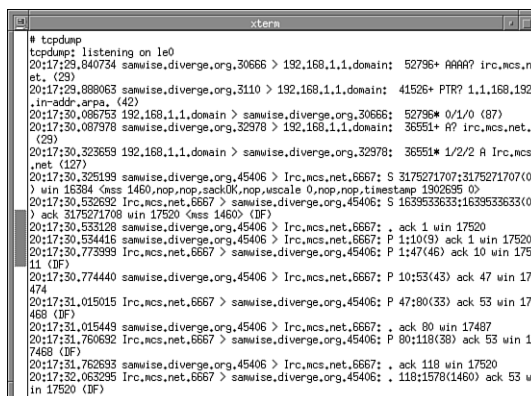
```
$ traceroute www.diverge.org
traceroute to www.diverge.org (192.168.2.2), 64 hops max, 40 byte packets
1 strider.diverge.org (192.168.1.1) 7.791 ms 7.182 ms 6.457 ms
2 gandalf.diverge.org (192.168.2.1) 43.978 ms 41.325 ms 43.904 ms
3 www.diverge.org (192.168.2.2) 41.293 ms 41.366 ms 41.683 ms
$
```

In this output, you easily can see the routers between my localhost and the remote host `www.diverge.org`—they are `strider.diverge.org` and `gandalf.diverge.org`. The fields are hop-number hostname IPaddress times.

Using the Sniffer `tcpdump`

A sniffer is a network-monitoring system that captures a great deal of information about the real content of a given network. Sniffers are particularly powerful because they allow for the storage of several layers of TCP information, which, of course, can be used for performing malicious attacks as well as monitoring.

On the upside, a sniffer can provide detailed information for troubleshooting efforts. Figure 3.4 is a snapshot of `tcpdump`.



```

# tcpdump
tcpdump: listening on le0
20:17:29.840734 sanwise,diverge.org,30666 > 192.168.1.1.domain: 52796* AAAA? irc.mcs.net. (29)
20:17:29.888063 sanwise,diverge.org,3110 > 192.168.1.1.domain: 41526* PTR? 1.1.168.192.in-addr.arpa. (42)
20:17:30.086753 192.168.1.1.domain > sanwise,diverge.org,30666: 52796* 0/1/0 (87)
20:17:30.087978 sanwise,diverge.org,32978 > 192.168.1.1.domain: 36551* A? irc.mcs.net. (29)
20:17:30.323659 192.168.1.1.domain > sanwise,diverge.org,32978: 36551* 1/2/2 A irc.mcs.net. (127)
20:17:30.325199 sanwise,diverge.org,45406 > irc.mcs.net,6667: S 3175271707:3175271707(0) win 15384 <msg 1460,nop,nop,sackOK,nop,wscale 0,nop,nop,timestamp 1902695 0>
20:17:30.532692 irc.mcs.net,6667 > sanwise,diverge.org,45406: S 1639533633:1639533633(0) ack 3175271708 win 17520 <msg 1460> (DF)
20:17:30.533128 sanwise,diverge.org,45406 > irc.mcs.net,6667: . ack 1 win 17520
20:17:30.534416 sanwise,diverge.org,45406 > irc.mcs.net,6667: P 1:10(9) ack 1 win 17520
20:17:30.773999 irc.mcs.net,6667 > sanwise,diverge.org,45406: P 1:47(46) ack 10 win 17511 (DF)
20:17:30.774440 sanwise,diverge.org,45406 > irc.mcs.net,6667: P 10:53(43) ack 47 win 17474
20:17:31.015015 irc.mcs.net,6667 > sanwise,diverge.org,45406: P 47:80(33) ack 53 win 17468 (DF)
20:17:31.015449 sanwise,diverge.org,45406 > irc.mcs.net,6667: . ack 80 win 17487
20:17:31.760692 irc.mcs.net,6667 > sanwise,diverge.org,45406: P 80:118(38) ack 53 win 17468 (DF)
20:17:31.762693 sanwise,diverge.org,45406 > irc.mcs.net,6667: . ack 118 win 17520
20:17:32.063295 irc.mcs.net,6667 > sanwise,diverge.org,45406: . 118:1578(1480) ack 53 win 17520 (DF)

```

FIGURE 3.4

tcpdump output.

The output when `tcpdump` is started with no options is dizzying. The `tcpdump` sniffer has an immense amount of command-line switches and options.

Some Practical Applications for `tcpdump`

Obviously, the output of `tcpdump` is rather intense. After reviewing some of the options, you have probably surmised that there are ways to focus the `tcpdump` utility to look for particular information. Table 3.11 describes some options and shows what particular problem they can be used to address.

TABLE 3.11

Command String	Application
tcpdump tcp host <ip_address>	SYN floods
tcpdump dst host <ip_address>	Network stack overflows (a.k.a. Ping of Death)
tcpdump icmp dst host <broadcast_address>	Smurf attacks
tcpdump -e host <ip_address>	Duplicate IP addresses
tcpdump arp	ARP misconfiguration
tcpdump icmp	Routing issues

As an example, let’s say that you want to look at ICMP traffic on the local network:

```
#
[root@kerry /root]# tcpdump icmp
Kernel filter, protocol ALL, datagram packet socket
tcpdump: listening on all devices
11:48:30.214757 eth1 M 172.16.141.99 > 192.187.225.50: icmp: echo request
11:48:30.215135 eth1 M 172.16.141.99 > arouter.local.net: icmp: echo request
11:48:32.277764 eth1 M 172.16.141.99 > frame.local.net: icmp: echo request
. . .
```

This output tells you that there are a lot of echo requests from the node at 172.16.141.99. In fact, this is a new router being installed, and the administrator is probing the network from the router.

ntop

One particularly interesting tool for constant network monitoring is the ntop utility. Basically, ntop displays the top network users. Figure 3.5 shows ntop running in an xterminal.

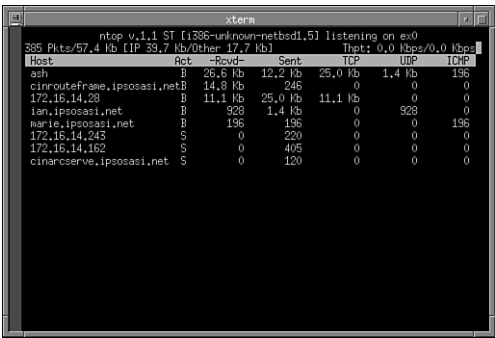


FIGURE 3.5
ntop in action.

The `ntop` utility has a vast array of command-line options and interactive commands. Table 3.12 gives a few examples of the more interesting ones.

TABLE 3.12 `ntop` Options

-r	Changes the rate that <code>ntop</code> updates the screen display. This is very helpful for determining time ranges in which a problem may be occurring.
-p	Specifies the IP protocol to monitor. Because the default is <code>all</code> , this option can act as a filter.
-l	Logs information captured by <code>ntop</code> into <code>ntop.log</code> . An application here is post-analysis of <code>ntop</code> results.

Interpreting `ntop` is pretty straightforward. The Host field contains either a hostname (if it can be resolved) or an IP address.

TABLE 3.13 `ntop` Output Fields

<i>Field</i>	<i>Description</i>
Host	Contains either a hostname (if it can be resolved) or an IP address
Act	Gives more information about the host: B indicates that a host has received and sent data. R indicates that a host has received data. S indicates that a host has sent data. I indicates that a host is idle.
Rcvd	Shows the amount of traffic that a host received between updates.
Sent	Shows the amount of traffic that a host sent between updates.
<protocol>	Gives three columns (TCP, UDP, and ICMP) that show the changes of the protocol type.

To make a little more sense of this, consider the following line as an example:

```
cingwise.ipsosasi.net  S  2.2 Kb  4.8 MB  1.8 Kb  420  0  0
```

The hostname is `cingwise.ipsosasi.net`, and the last thing the host did was send traffic. During the last update, it received 2.2Kb and sent 4.8MB; there was a difference of 420 bytes between updates in traffic with the TCP protocol.

The `ntop` utility is very useful for watching network activity in general.

Summary

This chapter examined popular Unix tools that have been ported to or rewritten for use on Linux. It also covered analyzing the output of these tools. The next chapter moves forward to tools that were specifically designed on Linux for performance monitoring.

