4

Performance Tools: Process-Specific CPU

After using the system-wide performance tools to figure out which process is slowing down the system, you must apply the process-specific performance tools to figure out how the process is behaving. Linux provides a rich set of tools to track the important statistics of a process and application's performance.

After reading this chapter, you should be able to

- Determine whether an application's runtime is spent in the kernel or application.
- Determine what library and system calls an application is making and how long they are taking.
- Profile an application to figure out what source lines and functions are taking the longest time to complete.

4.1 Process Performance Statistics

The tools to analyze the performance of applications are varied and have existed in one form or another since the early days of UNIX. It is critical to understand how an application is interacting with the operating system, CPU, and memory system to understand its performance. Most applications are not self-contained and make many calls to the Linux kernel and different libraries. These calls to the Linux kernel (or system calls) may be as

simple as "what's my PID?" or as complex as "read 12 blocks of data from the disk." Different systems calls will have different performance implications. Correspondingly, the library calls may be as simple as memory allocation or as complex as graphics window creation. These library calls may also have different performance characteristics.

4.1.1 Kernel Time Versus User Time

The most basic split of where an application may spend its time is between kernel and user time. Kernel time is the time spent in the Linux kernel, and user time is the amount of time spent in application or library code. Linux has tools such `time` and `ps` that can indicate (appropriately enough) whether an application is spending its time in application or kernel code. It also has commands such as `oprofile` and `strace` that enable you to trace which kernel calls are made on the behalf of the process, as well as how long each of those calls took to complete.

4.1.2 Library Time Versus Application Time

Any application with even a minor amount of complexity relies on system libraries to perform complex actions. These libraries may cause performance problems, so it is important to be able to see how much time an application spends in a particular library. Although it might not always be practical to modify the source code of the libraries directly to fix a problem, it may be possible to change the application code to call different or fewer library functions. The `ltrace` command and `oprofile` suite provide a way to analyze the performance of libraries when they are used by applications. Tools built in to the Linux loader, `ld`, helps you determine whether the use of many libraries slows down an application's start time.

4.1.3 Subdividing Application Time

When the application is known to be the bottleneck, Linux provides tools that enable you to profile an application to figure out where time is spent within an application. Tools such as `gprof` and `oprofile` can generate profiles of an application that pin down exactly which source line is causing large amounts of time to be spent.

4.2 The Tools

Linux has a variety of tools to help you determine which pieces of an application are the primary users of the CPU. This section describes these tools.

4.2.1 time

The `time` command performs a basic function when testing a command's performance, yet it is often the first place to turn. The `time` command acts as a stopwatch and times how long a command takes to execute. It measures three types of time. First, it measures the real or elapsed time, which is the amount of time between when the program started and finished execution. Next, it measures the user time, which is the amount of time that the CPU spent executing application code on behalf of the program. Finally, `time` measures system time, which is the amount of time the CPU spent executing system or kernel code on behalf of the application.

4.2.1.1 CPU Performance-Related Options

The `time` command (see Table 4-1) is invoked in the following manner:

```
time [-v] application
```

The `application` is timed, and information about its CPU usage is displayed on standard output after it has completed.

Table 4-1 *`time`* Command-Line Options

Option	Explanation
`-v`	This option presents a verbose display of the program's time and statistics. Some statistics are zeroed out, but more statistics are valid with Linux kernel v2.6 than with Linux kernel v2.4. Most of the valid statistics are present in both the standard and verbose mode, but the verbose mode provides a better description for each statistic.

Table 4-2 describes the valid output statistic that the `time` command provides. The rest are not measured and always display zero.

Table 4-2 CPU-Specific *`time`* Output

Column	Explanation
User time (seconds)	This is the number of seconds of CPU spent by the application.
System time (seconds)	This is the number of seconds spent in the Linux kernel on behalf of the application.
Elapsed (wall-clock) time (h:mm:ss or m:ss)	This is the amount of time elapsed (in wall-clock time) between when the application was launched and when it completed.
Percent of CPU this job got	This is the percentage of the CPU that the process consumed as it was running.
Major (requiring I/O) page faults	The number of major page faults or those that required a page of memory to be read from disk.
Minor (reclaiming a frame) page faults	The number of minor page faults or those that could be filled without going to disk.
Swaps	This is the number of times the process was swapped to disk.
Voluntary context switches	The number of times the process yielded the CPU (for example, by going to sleep).
Involuntary context switches:	The number of times the CPU was taken from the process.
Page size (bytes)	The page size of the system.
Exit status	The exit status of the application.

This command is a good way to start an investigation. It displays how long the application is taking to execute and how much of that time is spent in the Linux kernel versus your application.

4.2.1.2 Example Usage

The `time` command included on Linux is a part of the cross-platform GNU tools. The default command output prints a host of statistics about the commands run, even if Linux does not support them. If the statistics are not available, `time` just prints a zero. The following command is a simple invocation of the `time` command. You can see in Listing 4.1 that the elapsed time (~3 seconds) is much greater than the sum of the user (0.9 seconds) and system (0.13 seconds) time, because the application spends most of its time waiting for input and little time using the processor.

Listing 4.1

```
[ezolt@wintermute manuscript]$ /usr/bin/time gcalctool
0.91user 0.13system 0:03.37elapsed 30%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (2085major+369minor)pagefaults 0swaps
```

Listing 4.2 is an example of `time` displaying verbose output. As you can see, this output shows much more than the typical output of `time`. Unfortunately, most of the statistics are zeros, because they are not supported on Linux. For the most part, the information provided in verbose mode is identical to the output provided in standard mode, but the statistics' labels are much more descriptive. In this case, we can see that this process used 15 percent of the CPU when it was running, and spent 1.15 seconds running user code with .12 seconds running kernel code. It accrued 2,087 major page faults, or memory faults that did not require a trip to disk; it accrued 371 page faults that did require a trip to disk. A high number of major faults would indicate that the operating system was constantly going to disk when the application tried to use memory, which most likely means that the kernel was swapping a significant amount.

Listing 4.2

```
[ezolt@wintermute manuscript]$ /usr/bin/time --verbose gcalctool
        Command being timed: "gcalctool"
        User time (seconds): 1.15
        System time (seconds): 0.12
        Percent of CPU this job got: 15%
```

continues

Listing 4.2 (Continued)

```
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:08.02
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 0
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 2087
        Minor (reclaiming a frame) page faults: 371
        Voluntary context switches: 0
        Involuntary context switches: 0
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Note that the bash shell has a built-in `time` command, so if you are running bash and execute `time` without specifying the path to the executable, you get the following output:

```
[ezolt@wintermute manuscript]$ time gcalctool

real    0m3.409s
user    0m0.960s
sys     0m0.090s
```

The bash built-in `time` command can be useful, but it provides a subset of the process execution information.

4.2.2 strace

`strace` is a tool that traces the system calls that a program makes while executing. System calls are function calls made into the Linux kernel by or on behalf of an application. `strace` can show the exact system calls that were made and proves incredibly useful to determine how an application is using the Linux kernel. Tracing down the frequency and length of system calls can be especially valuable when analyzing a large program or one you do not understand completely. By looking at the `strace` output, you can get a feel for how the application is using the kernel and what type of functions it depends on.

`strace` can also be useful when you completely understand an application, but if that application makes calls to system libraries (such as `libc` or `GTK`.) In this case, even though you know where the application makes every system call, the libraries might be making more system calls on behalf of your application. `strace` can quickly show you what calls these libraries are making.

Although `strace` is mainly intended to trace the interaction of processes with the kernel by showing the arguments and results for every system call an application makes, `strace` can also provide summary information that is a little less daunting. After the run of an application, `strace` can provide a table showing the frequency of each system call and the total time spent in calls of that type. This table can be a crucial first piece of information in understanding how your program is interacting with the Linux kernel.

4.2.2.1 CPU Performance-Related Options

The following invocation of `strace` is most useful for performance testing:

```
strace [-c] [-p pid] [-o file] [--help] [ command [ arg ...  ]]
```

If `strace` is run without any options, it displays all the system calls made by the given command on standard error. This can be helpful when trying to figure out why an application is spending a large amount of time in the kernel. Table 4-3 describes a few `strace` options that are also helpful when tracing a performance problem.

Table 4-3 *strace* Command-Line Options

Option	Explanation
`-c`	This causes `strace` to print out a summary of statistics rather than an individual list of all the system calls that are made.
`-p pid`	This attaches to the process with the given PID and starts tracing.
`-o file`	The output of `strace` will be saved in `file`.
`--help`	Lists a complete summary of the `strace` options

Table 4-4 explains the statistics present in output of the `strace` summary option. Each line of output describes a set of statistics for a particular system call.

Table 4-4 CPU-Specific *strace* Output

Column	Explanation
`% time`	Of the total time spent making system calls, this is the percentage of time spent on this one.
`seconds`	This the total number of seconds spent in this system call.
`usecs/call`	This is the number of microseconds spent per system call of this type.
`calls`	This is the total number of calls of this type.
`errors`	This is the number of times that this system call returned an error.

Although the options just described are most relevant to a performance investigation, `strace` can also filter the types of system calls that it traces. The options to select the system calls to trace are described in detail with the `--help` option and in the `strace` man page. For general performance tuning, it is usually not necessary to use them; if needed, however, they exist.

4.2.2.2 Example Usage

Listing 4.3 is an example of using `strace` to gather statistics about which system calls an application is making. As you can see, `strace` provides a nice profile of the system's calls made on behalf of an application, which, in this case, is `oowriter`. In this example, we look at how `oowriter` is using the read system call. We can see that read is taking the 20 percent of the time by consuming a total of 0.44 seconds. It is called 2,427 times and, on average, each call takes 184 microseconds. Of those calls, 26 return an error.

Listing 4.3

```
[ezolt@wintermute tmp]$ strace -c oowriter
execve("/usr/bin/oowriter", ["oowriter"], [/* 35 vars */]) = 0
Starting OpenOffice.org ...

% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 20.57    0.445636         184      2427        26 read
 18.25    0.395386         229      1727           write
 11.69    0.253217         338       750       514 access
 10.81    0.234119       16723        14         6 waitpid
  9.53    0.206461        1043       198           select
  4.73    0.102520         201       511        55 stat64
  4.58    0.099290         154       646           gettimeofday
  4.41    0.095495          58      1656        15 lstat64
  2.51    0.054279         277       196           munmap
  2.32    0.050333         123       408           close
  2.07    0.044863          66       681       297 open
  1.98    0.042879         997        43           writev
  1.18    0.025614          12      2107           lseek
  0.95    0.020563        1210        17           unlink
  0.67    0.014550         231        63           getdents64
  0.58    0.012656          44       286           mmap2
  0.53    0.011399          68       167         2 ioctl
  0.50    0.010776         203        53           readv
  0.44    0.009500        2375         4         3 mkdir
```

continues

Listing 4.3 (Continued)

```
0.33    0.007233         603          12           clone
0.29    0.006255          28         224           old_mmap
0.24    0.005240        2620           2           vfork
0.24    0.005173          50         104           rt_sigprocmask
0.11    0.002311           8         295           fstat64
```

`strace` does a good job of tracking a process, but it does introduce some overhead when it is running on an application. As a result, the number of calls that `strace` reports is probably more reliable than the amount of time that it reports for each call. Use the times provided by `strace` as a starting point for investigation rather than a highly accurate measurement of how much time was spent in each call.

4.2.3 ltrace

`ltrace` is similar in concept to `strace`, but it traces the calls that an application makes to libraries rather than to the kernel. Although it is primarily used to provide an exact trace of the arguments and return values of library calls, you can also use `ltrace` to summarize how much time was spent in each call. This enables you to figure out both what library calls the application is making and how long each is taking.

Be careful when using `ltrace` because it can generate misleading results. The time spent may be counted twice if one library function calls another. For example, if library function foo() calls function bar(), the time reported for function foo() will be all the time spent running the code in function foo() plus all the time spent in function bar().

With this caveat in mind, it still is a useful tool to figure out how an application is behaving.

4.2.3.1 CPU Performance-Related Options

`ltrace` provides similar functionality to `strace` and is invoked in a similar way:

```
ltrace [-c] [-p pid] [-o filename] [-S] [--help] command
```

In the preceding invocation, `command` is the command that you want `ltrace` to trace. Without any options to `ltrace`, it displays all the library calls to standard error. Table 4-5 describes the `ltrace` options that are most relevant to a performance investigation.

Table 4-5 *`ltrace`* Command-Line Options

Option	Explanation
`-c`	This option causes `ltrace` to print a summary of all the calls after the command has completed.
`-S`	`ltrace` traces system calls in addition to library calls, which is identical to the functionality `strace` provides.
`-p pid`	This traces the process with the given PID.
`-o file`	The output of `strace` is saved in `file`.
`--help`	Displays help information about `ltrace`.

Again, the summary mode provides performance statistics about the library calls made during an application's execution. Table 4-6 describes the meanings of these statistics.

Table 4-6 CPU-Specific *`ltrace`* Output

Column	Explanation
`% time`	Of the total time spent making library calls, this is the percentage of time spent on this one.
`seconds`	This is the total number of seconds spent in this library call.
`usecs/call`	This is the number of microseconds spent per library call of this type.
`calls`	This is the total number of calls of this type.
`function`	This is the name of the library call.

Much like `strace`, `ltrace` has a large number of options that can modify the functions that it traces. These options are described by the `ltrace --help` command and in detail in the `ltrace` man page.

4.2.3.2 Example Usage

Listing 4.4 is a simple example of `ltrace` running on the `xeyes` command. `xeyes` is an X Window application that pops up a pair of eyes that follow your mouse pointer around the screen.

Listing 4.4

```
[ezolt@localhost manuscript]$ ltrace -c /usr/X11R6/bin/xeyes
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 18.65    0.065967       65967         1 XSetWMProtocols
 17.19    0.060803          86       702 hypot
 12.06    0.042654         367       116 XQueryPointer
  9.51    0.033632       33632         1 XtAppInitialize
  8.39    0.029684          84       353 XFillArc
  7.13    0.025204         107       234 cos
  6.24    0.022091          94       234 atan2
  5.56    0.019656          84       234 sin
  4.62    0.016337         139       117 XtAppAddTimeOut
  3.19    0.011297          95       118 XtWidgetToApplicationContext
  3.06    0.010827          91       118 XtWindowOfObject
  1.39    0.004934        4934         1 XtRealizeWidget
  1.39    0.004908        2454         2 XCreateBitmapFromData
  0.65    0.002291        2291         1 XtCreateManagedWidget
  0.12    0.000429         429         1 XShapeQueryExtension
  0.09    0.000332         332         1 XInternAtom
  0.09    0.000327          81         4 XtDisplay
  0.09    0.000320         106         3 XtGetGC
  0.05    0.000168          84         2 XSetForeground
  0.05    0.000166          83         2 XtScreen
```

```
  0.04    0.000153         153         1 XtParseTranslationTable
  0.04    0.000138         138         1 XtSetValues
  0.04    0.000129         129         1 XmuCvtStringToBackingStore
  0.03    0.000120         120         1 XtDestroyApplicationContext
  0.03    0.000116         116         1 XtAppAddActions
  0.03    0.000109         109         1 XCreatePixmap
  0.03    0.000108         108         1 XtSetLanguageProc
  0.03    0.000104         104         1 XtOverrideTranslations
  0.03    0.000102         102         1 XtWindow
  0.03    0.000096          96         1 XtAddConverter
  0.03    0.000093          93         1 XtCreateWindow
  0.03    0.000093          93         1 XFillRectangle
  0.03    0.000089          89         1 XCreateGC
  0.03    0.000089          89         1 XShapeCombineMask
  0.02    0.000087          87         1 XclearWindow
  0.02    0.000086          86         1 XFreePixmap
------ ----------- ----------- --------- --------------------
100.00    0.353739                  2261 total
```

In Listing 4.4, the library functions `XSetWMProtocols`, `hypot`, and `XQueryPointer` take 18.65 percent, 17.19 percent, and 12.06 percent of the total time spent in libraries. The call to the second most time-consuming function, `hypot`, is made 702 times, and the call to most time-consuming function, `XSetWMProtocols`, is made only once. Unless our application can completely remove the call to `XSetWMProtocols`, we are likely stuck with whatever time it takes. It is best to turn our attention to `hypot`. Each call to this function is relatively lightweight; so if we can reduce the number of times that it is called, we may be able to speed up the application. `hypot` would probably be the first function to be investigated if the `xeyes` application was a performance problem. Initially, we would determine what `hypot` does, but it is unclear where it may be documented. Possibly, we could figure out which library `hypot` belongs to and read the documentation for that library. In this case, we do not have to find the library first, because a man page exists for the `hypot` function. Running `man hypot` tells us that the `hypot` function will calculate the distance (hypotenuse) between two points and is part of the math library, `libm`. However, functions in libraries may have no man pages, so we would need to be able to

determine what library a function is part of without them. Unfortunately, `ltrace` does not make it at obvious which library a function is from. To figure it out, we have to use the Linux tools `ldd` and `objdump`. First, `ldd` is used to display which libraries are used by a dynamically linked application. Then, `objdump` is used to search each of those libraries for the given function. In Listing 4.5, we use `ldd` to see which libraries are used by the `xeyes` application.

Listing 4.5

```
[ezolt@localhost manuscript]$ ldd /usr/X11R6/bin/xeyes
        linux-gate.so.1 =>  (0x00ed3000)
        libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6 (0x00cd4000)
        libXt.so.6 => /usr/X11R6/lib/libXt.so.6 (0x00a17000)
        libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x00368000)
        libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x0034f000)
        libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x0032c000)
        libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x00262000)
        libm.so.6 => /lib/tls/libm.so.6 (0x00237000)
        libc.so.6 => /lib/tls/libc.so.6 (0x0011a000)
        libdl.so.2 => /lib/libdl.so.2 (0x0025c000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00101000)
```

Now that the `ldd` command has shown the libraries that `xeyes` uses, we can use the `objdump` command to figure out which library the function is in. In Listing 4.6, we look for the `hypot` symbol in each of the libraries that `xeyes` is linked to. The `-T` option of `objdump` lists all the symbols (mostly functions) that the library relies on or provides. By using `fgrep` to look at output lines that have `.text` in it, we can see which libraries export the `hypot` function. In this case, we can see that the `libm` library is the only library that contains the `hypot` function.

Listing 4.6

```
[/tmp]$ objdump -T /usr/X11R6/lib/libXmu.so.6 | fgrep ".text" | grep "hypot"
[/tmp]$ objdump -T /usr/X11R6/lib/libXt.so.6  | fgrep ".text" | grep "hypot"
[/tmp]$ objdump -T /usr/X11R6/lib/libSM.so.6  | fgrep ".text" | grep "hypot"
```

```
[/tmp]$ objdump -T /usr/X11R6/lib/libICE.so.6  | fgrep ".text" | grep "hypot"
[/tmp]$ objdump -T /usr/X11R6/lib/libXext.so.6 | fgrep ".text" | grep "hypot"
[/tmp]$ objdump -T /usr/X11R6/lib/libX11.so.6  | fgrep ".text" | grep "hypot"
[/tmp]$ objdump -T  /lib/tls/libm.so.6         | fgrep ".text" | grep "hypot"
00247520  w   DF .text  000000a9  GLIBC_2.0    hypotf
0024e810  w   DF .text  00000097  GLIBC_2.0    hypotl
002407c0  w   DF .text  00000097  GLIBC_2.0    hypot
[/tmp]$ objdump -T  /lib/tls/libc.so.6         | fgrep ".text" | grep "hypot"
[/tmp]$ objdump -T  /lib/libdl.so.2            | fgrep ".text" | grep "hypot"
[/tmp]$ objdump -T  /lib/ld-linux.so.2         | fgrep ".text" | grep "hypot"
```

The next step might be to look through the source of `xeyes` to figure out where `hypot` is called and, if possible, reduce the number of calls made to it. An alternative solution is to look at the source of `hypot` and try to optimize the source code of the library.

By enabling you to investigate which library calls are taking a long time to complete, `ltrace` enables you to determine the cost of each library call that an application makes.

4.2.4 ps (Process Status)

`ps` is an excellent command to track a process's behavior as it runs.

It provides detailed static and dynamic statistics about currently running processes. `ps` provides static information, such as command name and PID, as well as dynamic information, such as current use of memory and CPU.

4.2.4.1 CPU Performance-Related Options

`ps` has many different options and can retrieve many different statistics about the state of a running application. The following invocations are those options most related to CPU performance and will show information about the given PID:

```
ps [-o etime,time,pcpu,command] [-u user] [-U user] [PID]
```

The command `ps` is probably one of the oldest and feature-rich commands to extract performance information, which can make its use overwhelming. By only looking at a

subset of the total functionality, it is much more manageable. Table 4-7 contains the options that are most relevant to CPU performance.

Table 4-7 *ps* Command-Line Options

Option	Explanation
`-o <statistic>`	This option enables you to specify exactly what process statistics you want to track. The different statistics are specified in a comma-separated list with no spaces.
	`etime` Statistic: Elapsed time is the amount of time since the program began execution.
	`time` Statistic: CPU time is the amount of system plus user time the process spent running on the CPU.
	`pcpu` Statistic: The percentage of CPU that the process is currently consuming.
	`command` Statistic: This is the command name.
	`-A` Shows statistics about all processes.
	`-u user` Shows statistics about all processes with this effective user ID.
	`-U user` Shows statistic about all processes with this user ID.

`ps` provides myriad different performance statistics in addition to CPU statistics, many of which, such as a process's memory usage, are discussed in subsequent chapters.

4.2.4.2 Example Usage

This example shows a test application that is consuming 88 percent of the CPU and has been running for 6 seconds, but has only consumed 5 seconds of CPU time:

```
[ezolt@wintermute tmp]$ ps -o etime,time,pcpu,cmd 10882
    ELAPSED     TIME %CPU CMD
      00:06 00:00:05 88.0 ./burn
```

In Listing 4.7, instead of investigating the CPU performance of a particular process, we look at all the processes that a particular user is running. This may reveal information about the amount of resources a particular user consumes. In this case, we look at all the processes that the `netdump` user is running. Fortunately, `netdump` is a tame user and is only running `bash`, which is not taking up any of the CPU, and `top`, which is only taking up 0.5 percent of the CPU.

Listing 4.7

```
[/tmp]$ ps -o time,pcpu,command -u netdump
    TIME %CPU COMMAND
00:00:00  0.0 -bash
00:00:00  0.5 top
```

Unlike `time`, `ps` enables us to monitor information about a process currently running. For long-running jobs, you can use `ps` to periodically check the status of the process (instead of using it only to provide statistics about the program's execution after it has completed).

4.2.5 ld.so (Dynamic Loader)

When a dynamically linked application is executed, the Linux loader, `ld.so`, runs first. `ld.so` loads all the application's libraries and connects symbols that the application uses with the functions the libraries provide. Because different libraries were originally linked at different and possibly overlapping places in memory, the linker needs to sort through all the symbols and make sure that each lives at a different place in memory. When a symbol is moved from one virtual address to another, this is called a relocation. It takes time for the loader to do this, and it is much better if it does not need to be done at all. The prelink application aims to do that by rearranging the system libraries of the entire systems so that they do not overlap. An application with a high number of relocations may not have been prelinked.

The Linux loader usually runs without any intervention from the user, and by just executing a dynamic program, it is run automatically. Although the execution of the loader is hidden from the user, it still takes time to run and can potentially slow down an

application's startup time. When you ask for loader statistics, the loader shows the amount of work it is doing and enables you to figure out whether it is a bottleneck.

4.2.5.1 CPU Performance-Related Options

The `ld` command is invisibly run for every Linux application that uses shared libraries. By setting the appropriate environment variables, we can ask it to dump information about its execution. The following invocation influences `ld` execution:

```
env LD_DEBUG=statistics,help LD_DEBUG_OUTPUT=filename <command>
```

The debugging capabilities of the loader are completely controlled with environmental variables. Table 4-8 describes these variables.

Table 4-8 *ld* Environmental Variables

Option	Explanation
`LD_DEBUG=statistics`	This turns on the display of statistics for `ld`.
`LD_DEBUG=help`	This displays information about the available debugging statistics.

Table 4-9 describes some of the statistics that `ld.so` can provide. Time is given in clock cycles. To convert this to wall time, you must divide by the processor's clock speed. (This information is available from `cat /proc/cpuinfo`.)

Table 4-9 CPU Specific *ld.so* Output

Column	Explanation
`total startup time in dynamic loader`	The total amount of time (in clock cycles) spent in the load before the application started to execute.
`time needed for relocation`	The total amount of time (in clock cycles) spent relocating symbols.
`number of relocations`	The number of new relocation calculations done before the application's execution began.

Column	Explanation
`number of relocations from cache`	The number of relocations that were precalculated and used before the application started to execute.
`number of relative relocations`	The number of relative relocations.
`time needed to load objects`	The time needed to load all the libraries that an application is using.
`final number of relocations`	The total number of relocations made during an application run (including those made by `dlopen`).
`final number of relocations from cache`	The total number of relocations that were precalculated.

The information provided by `ld` can prove helpful in determining how much time is being spent setting up dynamic libraries before an application begins executing.

4.2.5.2 Example Usage

In Listing 4.8, we run an application with the `ld` debugging environmental variables defined. The output statistics are saved into the `lddebug` file. Notice that the loader shows two different sets of statistics. The first shows all the relocations that happen during startup, whereas the last shows all the statistics after the program closes. These can be different values if the application uses functions such as `dlopen`, which allows shared libraries to be mapped into an application after it has started to execute. In this case, we see that 83 percent of the time spent in the loader was doing allocations. If the application had been prelinked, this would have dropped close to zero.

Listing 4.8

```
[ezolt@wintermute ezolt]$ env LD_DEBUG=statistics LD_DEBUG_OUTPUT=lddebug gcalctool
[ezolt@wintermute ezolt]$ cat lddebug.2647
     2647:
     2647:     runtime linker statistics:
     2647:       total startup time in dynamic loader: 40820767 clock cycles
     2647:                 time needed for relocation: 33896920 clock cycles (83.0%)
     2647:                      number of relocations: 2821
```

continues

Listing 4.8 (Continued)

```
2647:          number of relocations from cache: 2284
2647:            number of relative relocations: 27717
2647:               time needed to load objects: 6421031 clock cycles (15.7%)
2647:
2647:     runtime linker statistics:
2647:               final number of relocations: 6693
2647:     final number of relocations from cache: 2284
```

If `ld` is determined to be the cause of sluggish startup time, it may be possible to reduce the amount of startup time by pruning the number of libraries that an application relies on or running `prelink` on the system.

4.2.6 gprof

A powerful way to profile applications on Linux is to use the `gprof` profiling command. `gprof` can show the call graph of application and sample where the application time is spent. `gprof` works by first instrumenting your application and then running the application to generate a sample file. `gprof` is very powerful, but requires application source and adds instrumentation overhead. Although it can accurately determine the number of times a function is called and approximate the amount of time spent in a function, the `gprof` instrumentation will likely change the timing characteristics of the application and slow down its execution.

4.2.6.1 CPU Performance-Related Options

To profile an application with `gprof`, you must have access to application source. You must then compile that application with a `gcc` command similar to the following:

```
gcc -gp -g3 -o app app.c
```

First, you must compile the application with profiling turned on, using `gcc`'s `-gp` option. You must take care not to strip the executable, and it is even more helpful to turn on symbols when compiling using the `-g3` option. Symbol information is

necessary to use the source annotation feature of `gprof`. When you run your instrumented application, an output file is generated. You can then use the `gprof` command to display the results. The `gprof` command is invoked as follows:

```
gprof [-p –flat-profile -q --graph --brief -A –annotated-source ] app
```

The options described in Table 4-10 specify what information gprof displays.

Table 4-10 *`gprof`* Command-Line Options

Option	Explanation
`--brief`	This option abbreviates the output of `gprof`. By default, `gprof` prints out all the performance information and a legend to describe what each metric means. This suppresses the legend.
`-p or --flat-profile`	This option prints out the total amount of time spent and the number of calls to each function in the application.
`-q or --graph`	This option prints out a call graph of the profiled application. This shows how the functions in the program called each other, how much time was spent in each of the functions, and how much was spent in the functions of the children.
`-A or --annotated-source`	This shows the profiling information next to the original source code.

Not all the output statistics are available for a particular profile. Which output statistic is available depends on how the application was compiled for profiling.

4.2.6.2 Example Usage

When profiling an application with `gprof`, the first step is to compile the application with profiling information. The compiler (`gcc`) inserts profiling information into the application and, when the application is run, it is saved into a file named `gmon.out`.

The burn test application is fairly simple. It clears a large area of memory and then calls two functions, `a()` and `b()`, which each touch this memory. Function `a()` touches the memory 10 times as often as function `b()`.

First, we compile the application:

```
[ezolt@wintermute test_app]$ gcc -pg -g3 -o burn_gprof burn.c
```

After we run in it, we can analyze the output. This is shown in Listing 4.9.

Listing 4.9

```
[ezolt@wintermute test_app]$ gprof  --brief -p ./burn_gprof
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 91.01      5.06     5.06        1     5.06     5.06  a
  8.99      5.56     0.50        1     0.50     0.50  b
```

In Listing 4.9, you can see `gprof` telling us what we already knew about the application. It has two functions, `a()` and `b()`. Each function is called once, and `a()` takes 10 times (91 percent) the amount of time to complete than `b()` (8.99 percent). 5.06 seconds of time is spent in the function `a()`, and .5 seconds is spent in function `b()`.

Listing 4.10 shows the call graph for the test application. The `<spontaneous>` comment listed in the output means that although `gprof` did not record any samples in `main()`, it deduced that `main()` must have been run, because functions `a()` and `b()` both had samples, and `main` was the only function in the code that called them. `gprof` most likely did not record any samples in `main()` because it is a very short function.

Listing 4.10

```
[ezolt@wintermute test_app]$ gprof  --brief -q ./burn_gprof
                     Call graph
```

```
granularity: each sample hit covers 4 byte(s) for 0.18% of 5.56 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    5.56                 main [1]
                5.06    0.00       1/1           a [2]
                0.50    0.00       1/1           b [3]
-----------------------------------------------
                5.06    0.00       1/1           main [1]
[2]     91.0    5.06    0.00       1         a [2]
-----------------------------------------------
                0.50    0.00       1/1           main [1]
[3]      9.0    0.50    0.00       1         b [3]
-----------------------------------------------

Index by function name

   [2] a                       [3] b
```

Finally, `gprof` can annotate the source code to show how often each function is called. Notice that Listing 4.11 does not show the time spent in the function; instead, it shows the number of times the function was called. As shown in the previous examples for `gprof`, `a()` actually took 10 times as long as `b()`, so be careful when optimizing. Do not assume that functions that are called a large number of times are actually using the CPU for a large amount of time or that functions that are called a few number of times are necessarily taking a small amount of the CPU time.

Listing 4.11

```
[ezolt@wintermute test_app]$ gprof -A burn_gprof
*** File /usr/src/perf/process_specific/test_app/burn.c:
                #include <string.h>

                #define ITER 10000
                #define SIZE 10000000
                #define STRIDE 10000
```

continues

Listing 4.11 (Continued)

```
                char test[SIZE];

                void a(void)
          1 -> {
                  int i=0,j=0;
                  for (j=0;j<10*ITER ; j++)
                    for (i=0;i<SIZE;i=i+STRIDE)
                      {
                        test[i]++;
                      }
                }

                void b(void)
          1 -> {
                  int i=0,j=0;
                  for (j=0;j<ITER; j++)
                    for (i=0;i<SIZE;i=i+STRIDE)
                      {
                        test[i]++;
                      }
                }


                main()
      ##### -> {

                  /* Arbitrary value*/
                  memset(test, 42, SIZE);
                  a();
                  b();
                }
```

```
Top 10 Lines:

     Line      Count

       10          1
       20          1

Execution Summary:

        3   Executable lines in this file
        3   Lines executed
   100.00   Percent of the file executed

        2   Total number of line executions
     0.67   Average executions per line
```

`gprof` provides a good summary of how many times functions or source lines in an application have been run and how long they took.

4.2.7 oprofile (II)

As discussed in Chapter 2, "Performance Tools: System CPU," you can use `oprofile` to track down the location of different events in the system or an application. `oprofile` is a lower-overhead tool than `gprof`. Unlike `gprof`, it does not require an application to be recompiled to be used. `oprofile` can also measure events not supported by `gprof`. Currently, `oprofile` can only support call graphs like those that `gprof` can generate with a kernel patch, whereas `gprof` can run on any Linux kernel.

4.2.7.1 CPU Performance-Related Options

The `oprofile` discussion in the section, "System-Wide Performance Tools" in Chapter 2 covers how to start profiling with `oprofile`. This section describes the parts of `oprofile` used to analyze the results of process-level sampling.

`oprofile` has a series of tools that display samples that have been collected. The first tool, `opreport`, displays information about how samples are distributed to the functions within executables and libraries. It is invoked as follows:

```
opreport [-d --details -f --long-filenames -l --symbols -l] application
```

Table 4-11 describes a few commands that can modify the level of information that `opreport` provides.

Table 4-11 *`opreport`* Command-Line Options

Option	Explanation
`-d or --details`	This shows an instruction-level breakdown of all the collected samples.
`-f or --long-filenames`	This shows the complete path name of the application being analyzed.
`-l or --symbols`	This shows how an application's samples are distributed to its symbols. This enables you to see what functions have the most samples attributed to them.

The next command that you can use to extract information about performance samples is `opannotate`. `opannotate` can attribute samples to specific source lines or assembly instructions. It is invoked as follows:

```
opannotate [-a  --assembly] [-s --source] application
```

The options described in Table 4-12 enable you to specify exactly what information `opannotate` will provide. One word of caution: because of limitations in the processor hardware counters at the source line and instruction level, the sample attributions may not be on the exact line that caused them; however, they will be near the actual event.

Table 4-12 *opannotate* Command-Line Options

Option	Explanation
`-s or --source --`	This shows the collected samples next to the application's source code.
`-a or --assembly`	This shows the samples collected next to the assembly code of the application.
`-s and -a`	If both `-s` and `-a` are specified, `opannotate` intermingles the source and assembly code with the samples.

When using `opannotate` and `opreport`, it is always best to specify the full path name to the application. If you do not, you may receive a cryptic error message (if `oprofile` cannot find the application's samples). By default, when displaying results, `opreport` only shows the executable name, which can be ambiguous in a system with multiple executables or libraries with identical names. Always specify the `-f` option so that `opreport` shows the complete path to the application.

`oprofile` also provides a command, `opgprof`, that can export the samples collected by `oprofile` into a form that `gprof` can digest. It is invoked in the following way:

```
opgprof application
```

This command takes the samples of `application` and generates a `gprof`-compatible profile. You can then view this file with the `gprof` command.

4.2.7.2 Example Usage

Because we already looked at `oprofile` in the section, "System-Wide Performance Tools" in Chapter 2, the examples here show you how to use `oprofile` to track down a performance problem to a particular line of source code. This section assumes that you have already started the profiling using the `opcontrol` command. The next step is to run the program that is having the performance problem. In this case, we use the burn program,

which is the same that we used in the gprof example. We start our test program as follows:

```
[ezolt@wintermute tmp]$ ./burn
```

After the program finishes, we must dump oprofile's buffers to disk, or else the samples will not be available to opreport. We do that using the following command:

```
[ezolt@wintermute tmp]$ sudo opcontrol -d
```

Next, in Listing 4.12 we ask opreport to tell us about the samples relevant to our test application, /tmp/burn. This gives us an overall view of how many cycles our application consumed. In this case, we see that 9,939 samples were taken for our application. As we dig into the oprofile tools, we will see how these samples are distributed within the burn application.

Listing 4.12

```
[ezolt@wintermute tmp]$ opreport -f /tmp/burn
CPU: PIII, speed 467.731 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a
unit mask of 0x00 (No unit mask) count 233865
     9939 100.0000 /tmp/burn
```

Next, in Listing 4.13, we want to see which functions in the burn application had all the samples. Because we are using the CPU_CLK_UNHALTED event, this roughly corresponds to the relative amount of time spent in each function. By looking at the output, we can see that 91 percent of the time was spent in function a(), and 9 percent was spent in function b().

Listing 4.13

```
[ezolt@wintermute tmp]$ opreport -l /tmp/burn
CPU: PIII, speed 467.731 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a
unit mask of 0x00 (No unit mask) count 233865
```

```
vma      samples  %           symbol name
08048348 9033     90.9118     a
0804839e 903       9.0882     b
```

In Listing 4.14, we ask `opreport` to show us which virtual addresses have samples attributed to them. In this case, it appears as if the instruction at address 0x0804838a has 75 percent of the samples. However, it is currently unclear what this instruction is doing or why.

Listing 4.14

```
[ezolt@wintermute tmp]$ opreport -d /tmp/burn
CPU: PIII, speed 467.731 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a
unit mask of 0x00 (No unit mask) count 233865
vma      samples  %           symbol name
08048348 9033     90.9118     a
 08048363 4         0.0443
 08048375 431       4.7714
 0804837c 271       3.0001
 0804837e 1         0.0111
 08048380 422       4.6718
 0804838a 6786     75.1245
 08048393 1114     12.3326
 08048395 4         0.0443
0804839e 903       9.0882     b
 080483cb 38        4.2082
 080483d2 19        2.1041
 080483d6 50        5.5371
 080483e0 697      77.1872
 080483e9 99       10.9635
```

Generally, it is more useful to know the source line that is using all the CPU time rather than the virtual address of the instruction that is using it. It is not always easy to

figure out the correspondence between a source line and a particular instruction; so, in Listing 4.15, we ask `opannotate` to do the hard work and show us the samples relative to the original source code (rather than the instruction's virtual address).

Listing 4.15

```
[ezolt@wintermute tmp]$ opannotate --source /tmp/burn
/*
 * Command line: opannotate --source /tmp/burn
 *
 * Interpretation of command line:
 * Output annotated source file with samples
 * Output all files
 *
 * CPU: PIII, speed 467.731 MHz (estimated)
 * Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with
a unit mask of 0x00 (No unit mask) count 233865
 */
/*
 * Total samples for file : "/tmp/burn.c"
 *
 *   9936 100.0000
 */


               :#include <string.h>
               :
               :#define ITER 10000
               :#define SIZE 10000000
               :#define STRIDE 10000
               :
               :char test[SIZE];
               :
               :void a(void)
               :{ /* a total:   9033 90.9118 */
```

```
             :  int i=0,j=0;
   8  0.0805 :  for (j=0;j<10*ITER ; j++)
8603 86.5841 :    for (i=0;i<SIZE;i=i+STRIDE)
             :      {
 422  4.2472 :        test[i]++;
             :      }
             :}
             :
             :void b(void)
             :{ /* b total:    903  9.0882 */
             :  int i=0,j=0;
             :  for (j=0;j<ITER; j++)
 853  8.5849 :    for (i=0;i<SIZE;i=i+STRIDE)
             :      {
  50  0.5032 :        test[i]++;
             :      }
             :}
             :
             :
             :main()
             :{
             :
             :  /* Arbitrary value*/
             :  memset(test, 42, SIZE);
             :  a();
             :  b();
             :}
```

As you can see in Listing 4.15, `opannotate` attributes most of the samples (86.59 percent) to the `for` loop in function `b()`. Unfortunately, this is a portion of the `for` loop that should not be expensive. Adding a fixed amount to an integer is very fast on modern processors, so the samples that `oprofile` reported were likely attributed to the wrong source line. The line below, `test[i]++;`, should be very expensive because it accesses the memory subsystem. This line is where the samples should have been attributed.

`oprofile` can mis-attribute samples for a few reasons beyond its control. First, the processor does not always interrupt on the exact line that caused the event to occur. This may cause samples to be attributed to instructions near the cause of the event, rather than to the exact instruction that caused the event. Second, when source code is compiled, compilers often rearrange instructions to make the executable more efficient. After a compiler has finished optimizing, code may not execute in the order that it was written. Separate source lines may have been rearranged and combined. As a result, a particular instruction may be the result of more than one source line, or may even be a compiler-generated intermediate piece of code that does not exist in the original source. As a result, when the compiler optimizes the code and generates machine instructions, there may no longer be a one-to-one mapping between the original lines of source code and the generated machine instructions. This can make it difficult or impossible for `oprofile` (and debuggers) to figure out exactly which line of source code corresponds to each machine instruction. However, `oprofile` tries to be as close as possible, so you can usually look a few lines above and below the line with the high sample count and figure out which code is truly expensive. If necessary, you can use `opannotate` to show the exact assembly instructions and virtual addresses that are receiving all the samples. It may be possible to figure out what the assembly instructions are doing and then map it back to your original source code by hand. `oprofile`'s sample attribution is not perfect, but it is usually close enough. Even with these limitations, the profiles provided by `oprofile` show the approximate source line to investigate, which is usually enough to figure out where the application is slowing down.

4.2.8 Languages: Static (C and C++) Versus Dynamic (Java and Mono)

The majority of the Linux performance tools support analysis of static languages such as C and C++, and all the tools described in this chapter work with applications written in these languages. The tools `ltrace`, `strace`, and `time` work with applications written in dynamic languages such as Java, Mono, Python, or Perl. However, the profiling tools `gprof` and `oprofile` cannot be used with these types of applications. Fortunately, most dynamic languages provide non-Linux–specific profiling infrastructures that you can use to generate similar types of profiles.

For Java applications, if the `java` command is run with the `-Xrunhprof` command-line option, `-Xrunhprof` profiles the application. More details are available at `http://antprof.sourceforge.net/hprof.html`. For Mono applications, if the `mono` executable is passed the `--profile` flag, it profiles the application. More details are available at `http://www.go-mono.com/performance.html`. Perl and Python have similar profile functionality, with Perl's `Devel::DProf` described at `http://www.perl.com/pub/a/2004/06/25/profiling.html`, and Python's `profiler` described at `http://docs.python.org/lib/profile.html`, respectively.

4.3 Chapter Summary

This chapter covered how to track the CPU performance bottlenecks of individual processes. You learned to determine how an application was spending its time by attributing the time spent to the Linux kernel, system libraries, or even to the application itself. You also learned how to figure out which calls were made to the kernel and system libraries and how long each took to complete. Finally, you learned how to profile an application and determine the particular line of source code that was spending a large amount of time. After mastering these tools, you can start with an application that hogs the CPU and use these tools to find the exact functions that are spending all the time.

Subsequent chapters investigate how to find bottlenecks that are not CPU bound. In particular, you learn about the tools used to find I/O bottlenecks, such as a saturated disk or an overloaded network.