7

The */proc* File System

TRY INVOKING THE `mount` COMMAND WITHOUT ARGUMENTS—this displays the file systems currently mounted on your GNU/Linux computer. You'll see one line that looks like this:

```
none on /proc type proc (rw)
```

This is the special `/proc` *file system*. Notice that the first field, `none`, indicates that this file system isn't associated with a hardware device such as a disk drive. Instead, `/proc` is a window into the running Linux kernel. Files in the `/proc` file system don't correspond to actual files on a physical device. Instead, they are magic objects that behave like files but provide access to parameters, data structures, and statistics in the kernel. The "contents" of these files are not always fixed blocks of data, as ordinary file contents are. Instead, they are generated on the fly by the Linux kernel when you read from the file. You can also change the configuration of the running kernel by writing to certain files in the `/proc` file system.

Let's look at an example:

```
% ls -l /proc/version
-r--r--r--    1 root     root            0 Jan 17 18:09 /proc/version
```

Note that the file size is zero; because the file's contents are generated by the kernel, the concept of file size is not applicable. Also, if you try this command yourself, you'll notice that the modification time on the file is the current time.

What's in this file? The contents of /proc/version consist of a string describing the Linux kernel version number. It contains the version information that would be obtained by the uname system call, described in Chapter 8, "Linux System Calls," in Section 8.15, "uname," plus additional information such as the version of the compiler that was used to compile the kernel. You can read from /proc/version like you would any other file. For instance, an easy way to display its contents is with the cat command.

```
% cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com) (gcc version egcs-2.91.
66 19990314/Linux (egcs-1.1.2 release)) #1 Tue Mar 7 21:07:39 EST 2000
```

The various entries in the /proc file system are described extensively in the proc man page (Section 5). To view it, invoke this command:

```
% man 5 proc
```

In this chapter, we'll describe some of the features of the /proc file system that are most likely to be useful to application programmers, and we'll give examples of using them. Some of the features of /proc are handy for debugging, too.

If you're interested in exactly how /proc works, take a look at the source code in the Linux kernel sources, under /usr/src/linux/fs/proc/.

7.1 Extracting Information from */proc*

Most of the entries in /proc provide information formatted to be readable by humans, but the formats are simple enough to be easily parsed. For example, /proc/cpuinfo contains information about the system CPU (or CPUs, for a multiprocessor machine). The output is a table of values, one per line, with a description of the value and a colon preceding each value.

For example, the output might look like this:

```
% cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 5
model name      : Pentium II (Deschutes)
stepping        : 2
cpu MHz         : 400.913520
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
sep_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 mmx fxsr
bogomips        : 399.77
```

We'll describe the interpretation of some of these fields in Section 7.3.1, "CPU Information."

A simple way to extract a value from this output is to read the file into a buffer and parse it in memory using `sscanf`. Listing 7.1 shows an example of this. The program includes the function `get_cpu_clock_speed` that reads from `/proc/cpuinfo` into memory and extracts the first CPU's clock speed.

Listing 7.1 **(*clock-speed.c*) Extract CPU Clock Speed from */proc/cpuinfo***

```
#include <stdio.h>
#include <string.h>

/* Returns the clock speed of the system's CPU in MHz, as reported by
   /proc/cpuinfo.  On a multiprocessor machine, returns the speed of
   the first CPU.  On error returns zero.  */

float get_cpu_clock_speed ()
{
  FILE* fp;
  char buffer[1024];
  size_t bytes_read;
  char* match;
  float clock_speed;

  /* Read the entire contents of /proc/cpuinfo into the buffer.  */
  fp = fopen ("/proc/cpuinfo", "r");
  bytes_read = fread (buffer, 1, sizeof (buffer), fp);
  fclose (fp);
  /* Bail if read failed or if buffer isn't big enough.  */
  if (bytes_read == 0 || bytes_read == sizeof (buffer))
    return 0;
  /* NUL-terminate the text.  */
  buffer[bytes_read] = '\0';
  /* Locate the line that starts with "cpu MHz".  */
  match = strstr (buffer, "cpu MHz");
  if (match == NULL)
    return 0;
  /* Parse the line to extract the clock speed.  */
  sscanf (match, "cpu MHz : %f", &clock_speed);
  return clock_speed;
}


int main ()
{
  printf ("CPU clock speed: %4.0f MHz\n", get_cpu_clock_speed ());
  return 0;
}
```

Be aware, however, that the names, semantics, and output formats of entries in the /proc file system might change in new Linux kernel revisions. If you use them in a program, you should make sure that the program's behavior degrades gracefully if the /proc entry is missing or is formatted unexpectedly.

7.2 Process Entries

The /proc file system contains a directory entry for each process running on the GNU/Linux system. The name of each directory is the process ID of the corresponding process.[1] These directories appear and disappear dynamically as processes start and terminate on the system. Each directory contains several entries providing access to information about the running process. From these process directories the /proc file system gets its name.

Each process directory contains these entries:

- cmdline contains the argument list for the process. The cmdline entry is described in Section 7.2.2, "Process Argument List."
- cwd is a symbolic link that points to the current working directory of the process (as set, for instance, with the chdir call).
- environ contains the process's environment. The environ entry is described in Section 7.2.3, "Process Environment."
- exe is a symbolic link that points to the executable image running in the process. The exe entry is described in Section 7.2.4, "Process Executable."
- fd is a subdirectory that contains entries for the file descriptors opened by the process. These are described in Section 7.2.5, "Process File Descriptors."
- maps displays information about files mapped into the process's address. See Chapter 5, "Interprocess Communication," Section 5.3, "Mapped Memory," for details of how memory-mapped files work. For each mapped file, maps displays the range of addresses in the process's address space into which the file is mapped, the permissions on these addresses, the name of the file, and other information.

  The maps table for each process displays the executable running in the process, any loaded shared libraries, and other files that the process has mapped in.
- root is a symbolic link to the root directory for this process. Usually, this is a symbolic link to /, the system root directory. The root directory for a process can be changed using the chroot call or the chroot command.[2]

1. On some UNIX systems, the process IDs are padded with zeros. On GNU/Linux, they are not.

2. The chroot call and command are outside the scope of this book. See the chroot man page in Section 1 for information about the command (invoke man 1 chroot), or the chroot man page in Section 2 (invoke man 2 chroot) for information about the call.

- `stat` contains lots of status and statistical information about the process. These are the same data as presented in the `status` entry, but in raw numerical format, all on a single line. The format is difficult to read but might be more suitable for parsing by programs.

  If you want to use the `stat` entry in your programs, see the `proc` man page, which describes its contents, by invoking `man 5 proc`.
- `statm` contains information about the memory used by the process. The `statm` entry is described in Section 7.2.6, "Process Memory Statistics."
- `status` contains lots of status and statistical information about the process, formatted to be comprehensible by humans. Section 7.2.7, "Process Statistics," contains a description of the `status` entry.
- The `cpu` entry appears only on SMP Linux kernels. It contains a breakdown of process time (user and system) by CPU.

Note that for security reasons, the permissions of some entries are set so that only the user who owns the process (or the superuser) can access them.

7.2.1 */proc/self*

One additional entry in the `/proc` file system makes it easy for a program to use `/proc` to find information about its own process. The entry `/proc/self` is a symbolic link to the `/proc` directory corresponding to the current process. The destination of the `/proc/self` link depends on which process looks at it: Each process sees its own process directory as the target of the link.

For example, the program in Listing 7.2 reads the target of the `/proc/self` link to determine its process ID. (We're doing it this way for illustrative purposes only; calling the `getpid` function, described in Chapter 3, "Processes," in Section 3.1.1, "Process IDs," is a much easier way to do the same thing.) This program uses the `readlink` system call, described in Section 8.11, "`readlink`: Reading Symbolic Links," to extract the target of the symbolic link.

Listing 7.2 **(*get-pid.c*) Obtain the Process ID from */proc/self***

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

/* Returns the process ID of the calling processes, as determined from
   the /proc/self symlink.  */

pid_t get_pid_from_proc_self ()
{
  char target[32];
  int pid;
  /* Read the target of the symbolic link.  */
  readlink ("/proc/self", target, sizeof (target));
```

continues

Listing 7.2 **Continued**

```
  /* The target is a directory named for the process ID.  */
  sscanf (target, "%d", &pid);
  return (pid_t) pid;
}

int main ()
{
  printf ("/proc/self reports process id %d\n",
          (int) get_pid_from_proc_self ());
  printf ("getpid() reports process id %d\n", (int) getpid ());
  return 0;
}
```

7.2.2 Process Argument List

The `cmdline` entry contains the process argument list (see Chapter 2, "Writing Good GNU/Linux Software," Section 2.1.1, "The Argument List"). The arguments are presented as a single character string, with arguments separated by NULs. Most string functions expect that the entire character string is terminated with a single NUL and will not handle NULs embedded within strings, so you'll have to handle the contents specially.

> **NUL vs. NULL**
>
> NUL is the character with integer value 0. It's different from NULL, which is a pointer with value 0.
>
> In C, a character string is usually terminated with a NUL character. For instance, the character string `"Hello, world!"` occupies 14 bytes because there is an implicit NUL after the exclamation point indicating the end of the string.
>
> NULL, on the other hand, is a pointer value that you can be sure will never correspond to a real memory address in your program.
>
> In C and C++, NUL is expressed as the character constant `'\0'`, or `(char) 0`. The definition of NULL differs among operating systems; on Linux, it is defined as `((void*)0)` in C and simply `0` in C++.

In Section 2.1.1, we presented a program in Listing 2.1 that printed out its own argument list. Using the `cmdline` entries in the `/proc` file system, we can implement a program that prints the argument of another process. Listing 7.3 is such a program; it prints the argument list of the process with the specified process ID. Because there may be several NULs in the contents of `cmdline` rather than a single one at the end, we can't determine the length of the string with `strlen` (which simply counts the number of characters until it encounters a NUL). Instead, we determine the length of `cmdline` from `read`, which returns the number of bytes that were read.

Listing 7.3 (*print-arg-list.c*) **Print the Argument List of a Running Process**

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

/* Prints the argument list, one argument to a line, of the process
   given by PID.  */

void print_process_arg_list (pid_t pid)
{
  int fd;
  char filename[24];
  char arg_list[1024];
  size_t length;
  char* next_arg;

  /* Generate the name of the cmdline file for the process. */
  snprintf (filename, sizeof (filename), "/proc/%d/cmdline", (int) pid);
  /* Read the contents of the file. */
  fd = open (filename, O_RDONLY);
  length = read (fd, arg_list, sizeof (arg_list));
  close (fd);
  /* read does not NUL-terminate the buffer, so do it here. */
  arg_list[length] = '\0';

  /* Loop over arguments.  Arguments are separated by NULs. */
  next_arg = arg_list;
  while (next_arg < arg_list + length) {
    /* Print the argument.  Each is NUL-terminated, so just treat it
       like an ordinary string.  */
    printf ("%s\n", next_arg);
    /* Advance to the next argument.  Since each argument is
       NUL-terminated, strlen counts the length of the next argument,
       not the entire argument list.  */
    next_arg += strlen (next_arg) + 1;
  }
}

int main (int argc, char* argv[])
{
  pid_t pid = (pid_t) atoi (argv[1]);
  print_process_arg_list (pid);
  return 0;
}
```

For example, suppose that process 372 is the system logger daemon, `syslogd`.

```
% ps 372
  PID TTY      STAT   TIME COMMAND
  372 ?        S      0:00 syslogd -m 0
% ./print-arg-list 372
syslogd
-m
0
```

In this case, `syslogd` was invoked with the arguments `-m 0`.

7.2.3 Process Environment

The `environ` entry contains a process's environment (see Section 2.1.6, "The Environment"). As with `cmdline`, the individual environment variables are separated by NULs. The format of each element is the same as that used in the `environ` variable, namely *`VARIABLE=value`*.

Listing 7.4 presents a generalization of the program in Listing 2.3 in Section 2.1.6. This version takes a process ID number on its command line and prints the environment for that process by reading it from `/proc`.

Listing 7.4 **(*print-environment.c*) Display the Environment of a Process**

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

/* Prints the environment, one environment variable to a line, of the
   process given by PID.  */

void print_process_environment (pid_t pid)
{
  int fd;
  char filename[24];
  char environment[8192];
  size_t length;
  char* next_var;

  /* Generate the name of the environ file for the process. */
  snprintf (filename, sizeof (filename), "/proc/%d/environ", (int) pid);
  /* Read the contents of the file. */
  fd = open (filename, O_RDONLY);
  length = read (fd, environment, sizeof (environment));
  close (fd);
  /* read does not NUL-terminate the buffer, so do it here. */
  environment[length] = '\0';
```

```
  /* Loop over variables.  Variables are separated by NULs. */
  next_var = environment;
  while (next_var < environment + length) {
    /* Print the variable.  Each is NUL-terminated, so just treat it
       like an ordinary string.  */
    printf ("%s\n", next_var);
    /* Advance to the next variable.  Since each variable is
       NUL-terminated, strlen counts the length of the next variable,
       not the entire variable list.  */
    next_var += strlen (next_var) + 1;
  }
}

int main (int argc, char* argv[])
{
  pid_t pid = (pid_t) atoi (argv[1]);
  print_process_environment (pid);
  return 0;
}
```

7.2.4 Process Executable

The `exe` entry points to the executable file being run in a process. In Section 2.1.1, we explained that typically the program executable name is passed as the first element of the argument list. Note, though, that this is purely conventional; a program may be invoked with any argument list. Using the `exe` entry in the `/proc` file system is a more reliable way to determine which executable is running.

One useful technique is to extract the path containing the executable from the `/proc` file system. For many programs, auxiliary files are installed in directories with known paths relative to the main program executable, so it's necessary to determine where that executable actually is. The function `get_executable_path` in Listing 7.5 determines the path of the executable running in the calling process by examining the symbolic link `/proc/self/exe`.

Listing 7.5 **(*get-exe-path.c*) Get the Path of the Currently Running Program Executable**

```
#include <limits.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/* Finds the path containing the currently running program executable.
   The path is placed into BUFFER, which is of length LEN.  Returns
   the number of characters in the path, or -1 on error.  */
```

continues

Listing 7.5 **Continued**

```
size_t get_executable_path (char* buffer, size_t len)
{
  char* path_end;
  /* Read the target of /proc/self/exe.  */
  if (readlink ("/proc/self/exe", buffer, len) <= 0)
    return -1;
  /* Find the last occurrence of a forward slash, the path separator.  */
  path_end = strrchr (buffer, '/');
  if (path_end == NULL)
    return -1;
  /* Advance to the character past the last slash.  */
  ++path_end;
  /* Obtain the directory containing the program by truncating the
     path after the last slash.  */
  *path_end = '\0';
  /* The length of the path is the number of characters up through the
     last slash.  */
  return (size_t) (path_end - buffer);
}

int main ()
{
  char path[PATH_MAX];
  get_executable_path (path, sizeof (path));
  printf ("this program is in the directory %s\n", path);
  return 0;
}
```

7.2.5 Process File Descriptors

The `fd` entry is a subdirectory that contains entries for the file descriptors opened by a process. Each entry is a symbolic link to the file or device opened on that file descriptor. You can write to or read from these symbolic links; this writes to or reads from the corresponding file or device opened in the target process. The entries in the `fd` subdirectory are named by the file descriptor numbers.

Here's a neat trick you can try with `fd` entries in `/proc`. Open a new window, and find the process ID of the shell process by running `ps`.

```
% ps
  PID TTY          TIME CMD
 1261 pts/4    00:00:00 bash
 2455 pts/4    00:00:00 ps
```

In this case, the shell (`bash`) is running in process 1261. Now open a second window, and look at the contents of the `fd` subdirectory for that process.

```
% ls -l /proc/1261/fd
total 0
lrwx------    1 samuel   samuel         64 Jan 30 01:02 0 -> /dev/pts/4
lrwx------    1 samuel   samuel         64 Jan 30 01:02 1 -> /dev/pts/4
lrwx------    1 samuel   samuel         64 Jan 30 01:02 2 -> /dev/pts/4
```

(There may be other lines of output corresponding to other open file descriptors as well.) Recall that we mentioned in Section 2.1.4, "Standard I/O," that file descriptors 0, 1, and 2 are initialized to standard input, output, and error, respectively. Thus, by writing to `/proc/1261/fd/1`, you can write to the device attached to `stdout` for the shell process—in this case, the pseudo TTY in the first window. In the second window, try writing a message to that file:

```
% echo "Hello, world." >> /proc/1261/fd/1
```

The text appears in the first window.

File descriptors besides standard input, output, and error appear in the `fd` subdirectory, too. Listing 7.6 presents a program that simply opens a file descriptor to a file specified on the command line and then loops forever.

Listing 7.6 **(*open-and-spin.c*) Open a File for Reading**

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  const char* const filename = argv[1];
  int fd = open (filename, O_RDONLY);
  printf ("in process %d, file descriptor %d is open to %s\n",
          (int) getpid (), (int) fd, filename);
  while (1);
  return 0;
}
```

Try running it in one window:

```
%  ./open-and-spin /etc/fstab
in process 2570, file descriptor 3 is open to /etc/fstab
```

In another window, take a look at the `fd` subdirectory corresponding to this process in `/proc`.

```
% ls -l /proc/2570/fd
total 0
lrwx------    1 samuel   samuel         64 Jan 30 01:30 0 -> /dev/pts/2
```

```
lrwx------    1 samuel   samuel         64 Jan 30 01:30 1 -> /dev/pts/2
lrwx------    1 samuel   samuel         64 Jan 30 01:30 2 -> /dev/pts/2
lr-x------    1 samuel   samuel         64 Jan 30 01:30 3 -> /etc/fstab
```

Notice the entry for file descriptor 3, linked to the file `/etc/fstab` opened on this descriptor.

File descriptors can be opened on sockets or pipes, too (see Chapter 5 for more information about these). In such a case, the target of the symbolic link corresponding to the file descriptor will state "socket" or "pipe" instead of pointing to an ordinary file or device.

7.2.6 Process Memory Statistics

The `statm` entry contains a list of seven numbers, separated by spaces. Each number is a count of the number of pages of memory used by the process in a particular category. The categories, in the order the numbers appear, are listed here:

- The total process size
- The size of the process resident in physical memory
- The memory shared with other processes—that is, memory mapped both by this process and at least one other (such as shared libraries or untouched copy-on-write pages)
- The text size of the process—that is, the size of loaded executable code
- The size of shared libraries mapped into this process
- The memory used by this process for its stack
- The number of dirty pages—that is, pages of memory that have been modified by the program

7.2.7 Process Statistics

The `status` entry contains a variety of information about the process, formatted for comprehension by humans. Among this information is the process ID and parent process ID, the real and effective user and group IDs, memory usage, and bit masks specifying which signals are caught, ignored, and blocked.

7.3 Hardware Information

Several of the other entries in the `/proc` file system provide access to information about the system hardware. Although these are typically of interest to system configurators and administrators, the information may occasionally be of use to application programmers as well. We'll present some of the more useful entries here.

7.3.1 CPU Information

As shown previously, `/proc/cpuinfo` contains information about the CPU or CPUs running the GNU/Linux system. The Processor field lists the processor number; this is 0 for single-processor systems. The Vendor, CPU Family, Model, and Stepping fields enable you to determine the exact model and revision of the CPU. More useful, the Flags field shows which CPU flags are set, which indicates the features available in this CPU. For example, "mmx" indicates the availability of the extended MMX instructions.[3]

Most of the information returned from `/proc/cpuinfo` is derived from the `cpuid` x86 assembly instruction. This instruction is the low-level mechanism by which a program obtains information about the CPU. For a greater understanding of the output of `/proc/cpuinfo`, see the documentation of the `cpuid` instruction in Intel's *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. This manual is available from `http://developer.intel.com/design`.

The last element, `bogomips`, is a Linux-specific value. It is a measurement of the processor's speed spinning in a tight loop and is therefore a rather poor indicator of overall processor speed.

7.3.2 Device Information

The `/proc/devices` file lists major device numbers for character and block devices available to the system. See Chapter 6, "Devices," for information about types of devices and device numbers.

7.3.3 PCI Bus Information

The `/proc/pci` file lists a summary of devices attached to the PCI bus or buses. These are actual PCI expansion cards and may also include devices built into the system's motherboard, plus AGP graphics cards. The listing includes the device type; the device and vendor ID; a device name, if available; information about the features offered by the device; and information about the PCI resources used by the device.

7.3.4 Serial Port Information

The `/proc/tty/driver/serial` file lists configuration information and statistics about serial ports. Serial ports are numbered from 0.[4] Configuration information about serial ports can also be obtained, as well as modified, using the `setserial` command. However, `/proc/tty/driver/serial` displays additional statistics about each serial port's interrupt counts.

3. See the *IA-32 Intel Architecture Software Developer's Manual* for documentation about MMX instructions, and see Chapter 9, "Inline Assembly Code," in this book for information on how to use these and other special assembly instructions in GNU/Linux programs.

4. Note that under DOS and Windows, serial ports are numbered from 1, so `COM1` corresponds to serial port number 0 under Linux.

For example, this line from /proc/tty/driver/serial might describe serial port 1 (which would be COM2 under Windows):

```
1: uart:16550A port:2F8 irq:3 baud:9600 tx:11 rx:0
```

This indicates that the serial port is run by a 16550A-type UART, uses I/O port 0x2f8 and IRQ 3 for communication, and runs at 9,600 baud. The serial port has seen 11 transmit interrupts and 0 receive interrupts.

See Section 6.4, "Hardware Devices," for information about serial devices.

7.4 Kernel Information

Many of the entries in /proc provide access to information about the running kernel's configuration and state. Some of these entries are at the top level of /proc; others are under /proc/sys/kernel.

7.4.1 Version Information

The file /proc/version contains a long string describing the kernel's release number and build version. It also includes information about how the kernel was built: the user who compiled it, the machine on which it was compiled, the date it was compiled, and the compiler release that was used—for example:

```
% cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com) (gcc version
egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)) #1 Tue Mar 7
21:07:39 EST 2000
```

This indicates that the system is running a 2.2.14 release of the Linux kernel, which was compiled with EGCS release 1.1.2. (EGCS, the *Experimental GNU Compiler System*, was a precursor to the current GCC project.)

The most important items in this output, the OS name and kernel version and revision, are available in separate /proc entries as well. These are /proc/sys/kernel/ostype, /proc/sys/kernel/osrelease, and /proc/sys/kernel/version, respectively.

```
% cat /proc/sys/kernel/ostype
Linux
% cat /proc/sys/kernel/osrelease
2.2.14-5.0
% cat /proc/sys/kernel/version
#1 Tue Mar 7 21:07:39 EST 2000
```

7.4.2 Hostname and Domain Name

The /proc/sys/kernel/hostname and /proc/sys/kernel/domainname entries contain the computer's hostname and domain name, respectively. This information is the same as that returned by the uname system call, described in Section 8.15.

7.4.3 Memory Usage

The `/proc/meminfo` entry contains information about the system's memory usage. Information is presented both for physical memory and for swap space. The first three lines present memory totals, in bytes; subsequent lines summarize this information in kilobytes—for example:

```
% cat /proc/meminfo
        total:    used:    free:  shared: buffers:  cached:
Mem:  529694720 519610368 10084352 82612224 10977280 82108416
Swap: 271392768 44003328 227389440
MemTotal:    517280 kB
MemFree:       9848 kB
MemShared:    80676 kB
Buffers:      10720 kB
Cached:       80184 kB
BigTotal:         0 kB
BigFree:          0 kB
SwapTotal:   265032 kB
SwapFree:    222060 kB
```

This shows 512MB physical memory, of which about 9MB is free, and 258MB of swap space, of which 216MB is free. In the row corresponding to physical memory, three other values are presented:

- The Shared column displays total shared memory currently allocated on the system (see Section 5.1, "Shared Memory").
- The Buffers column displays the memory allocated by Linux for block device buffers. These buffers are used by device drivers to hold blocks of data being read from and written to disk.
- The Cached column displays the memory allocated by Linux to the page cache. This memory is used to cache accesses to mapped files.

You can use the `free` command to display the same memory information.

7.5 Drives, Mounts, and File Systems

The `/proc` file system also contains information about the disk drives present in the system and the file systems mounted from them.

7.5.1 File Systems

The `/proc/filesystems` entry displays the file system types known to the kernel. Note that this list isn't very useful because it is not complete: File systems can be loaded and unloaded dynamically as kernel modules. The contents of `/proc/filesystems` list only file system types that either are statically linked into the kernel or are currently loaded. Other file system types may be available on the system as modules but might not be loaded yet.

7.5.2 Drives and Partitions

The /proc file system includes information about devices connected to both IDE controllers and SCSI controllers (if the system includes them).

On typical systems, the /proc/ide subdirectory may contain either or both of two subdirectories, ide0 and ide1, corresponding to the primary and secondary IDE controllers on the system.[5] These contain further subdirectories corresponding to physical devices attached to the controllers. The controller or device directories may be absent if Linux has not recognized any connected devices. The full paths corresponding to the four possible IDE devices are listed in Table 7.1.

Table 7.1 **Full Paths Corresponding to the Four Possible IDE Devices**

Controller	Device	Subdirectory
Primary	Master	/proc/ide/ide0/hda/
Primary	Slave	/proc/ide/ide0/hdb/
Secondary	Master	/proc/ide/ide1/hdc/
Secondary	Slave	/proc/ide/ide1/hdd/

See Section 6.4, "Hardware Devices," for more information about IDE device names.

Each IDE device directory contains several entries providing access to identification and configuration information for the device. A few of the most useful are listed here:

- model contains the device's model identification string.
- media contains the device's media type. Possible values are disk, cdrom, tape, floppy, and UNKNOWN.
- capacity contains the device's capacity, in 512-byte blocks. Note that for CD-ROM devices, the value will be $2^{31} - 1$, not the capacity of the disk in the drive. Note that the value in capacity represents the capacity of the entire physical disk; the capacity of file systems contained in partitions of the disk will be smaller.

For example, these commands show how to determine the media type and device identification for the master device on the secondary IDE controller. In this case, it turns out to be a Toshiba CD-ROM drive.

```
% cat /proc/ide/ide1/hdc/media
cdrom
% cat /proc/ide/ide1/hdc/model
TOSHIBA CD-ROM XM-6702B
```

5. If properly configured, the Linux kernel can support additional IDE controllers. These are numbered sequentially from ide2.

If SCSI devices are present in the system, `/proc/scsi/scsi` contains a summary of their identification values. For example, the contents might look like this:

```
% cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: QUANTUM  Model: ATLAS_V__9_WLS   Rev: 0230
  Type:   Direct-Access                   ANSI SCSI revision: 03
Host: scsi0 Channel: 00 Id: 04 Lun: 00
  Vendor: QUANTUM  Model: QM39100TD-SW     Rev: N491
  Type:   Direct-Access                   ANSI SCSI revision: 02
```

This computer contains one single-channel SCSI controller (designated "scsi0"), to which two Quantum disk drives are connected, with SCSI device IDs 0 and 4.

The `/proc/partitions` entry displays the partitions of recognized disk devices. For each partition, the output includes the major and minor device number, the number of 1024-byte blocks, and the device name corresponding to that partition.

The `/proc/sys/dev/cdrom/info` entry displays miscellaneous information about the capabilities of CD-ROM drives. The fields are self-explanatory:

```
%  cat /proc/sys/dev/cdrom/info
CD-ROM information, Id: cdrom.c 2.56 1999/09/09

drive name:  hdc
drive speed:  48
drive # of slots:  0
Can close tray:  1
Can open tray:  1
Can lock tray:  1
Can change speed:  1
Can select disk:  0
Can read multisession:  1
Can read MCN:  1
Reports media changed:  1
Can play audio:  1
```

7.5.3 Mounts

The `/proc/mounts` file provides a summary of mounted file systems. Each line corresponds to a single *mount descriptor* and lists the mounted device, the mount point, and other information. Note that `/proc/mounts` contains the same information as the ordinary file `/etc/mtab`, which is automatically updated by the `mount` command.

These are the elements of a mount descriptor:

- The first element on the line is the mounted device (see Chapter 6). For special file systems such as the `/proc` file system, this is `none`.
- The second element is the *mount point*, the place in the root file system at which the file system contents appear. For the root file system itself, the mount point is listed as `/`. For swap drives, the mount point is listed as `swap`.

- The third element is the file system type. Currently, most GNU/Linux systems use the `ext2` file system for disk drives, but DOS or Windows drives may be mounted with other file system types, such as `fat` or `vfat`. Most CD-ROMs contain an `iso9660` file system. See the man page for the `mount` command for a list of file system types.
- The fourth element lists mount flags. These are options that were specified when the mount was added. See the man page for the `mount` command for an explanation of flags for the various file system types.

In `/proc/mounts`, the last two elements are always 0 and have no meaning.

See the man page for `fstab` for details about the format of mount descriptors.[6] GNU/Linux includes functions to help you parse mount descriptors; see the man page for the `getmntent` function for information on using these.

7.5.4 Locks

Section 8.3, "`fcntl`: Locks and Other File Operations," describes how to use the `fcntl` system call to manipulate read and write locks on files. The `/proc/locks` entry describes all the file locks currently outstanding in the system. Each row in the output corresponds to one lock.

For locks created with `fcntl`, the first two entries on the line are `POSIX ADVISORY`. The third is `WRITE` or `READ`, depending on the lock type. The next number is the process ID of the process holding the lock. The following three numbers, separated by colons, are the major and minor device numbers of the device on which the file resides and the *inode* number, which locates the file in the file system. The remainder of the line lists values internal to the kernel that are not of general utility.

Turning the contents of `/proc/locks` into useful information takes some detective work. You can watch `/proc/locks` in action, for instance, by running the program in Listing 8.2 to create a write lock on the file `/tmp/test-file`.

```
% touch /tmp/test-file
% ./lock-file /tmp/test-file
file /tmp/test-file
opening /tmp/test-file
locking
locked; hit enter to unlock...
```

In another window, look at the contents of `/proc/locks`.

```
% cat /proc/locks
1: POSIX  ADVISORY  WRITE 5467 08:05:181288 0 2147483647 d1b5f740 00000000
dfea7d40 00000000 00000000
```

6. The `/etc/fstab` file lists the static mount configuration of the GNU/Linux system.

There may be other lines of output, too, corresponding to locks held by other programs. In this case, 5467 is the process ID of the `lock-file` program. Use `ps` to figure out what this process is running.

```
% ps 5467
  PID TTY      STAT   TIME COMMAND
 5467 pts/28   S      0:00 ./lock-file /tmp/test-file
```

The locked file, `/tmp/test-file`, resides on the device that has major and minor device numbers 8 and 5, respectively. These numbers happen to correspond to `/dev/sda5`.

```
% df /tmp
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/sda5              8459764   5094292   2935736  63% /
% ls -l /dev/sda5
brw-rw----    1 root     disk       8,   5 May  5  1998 /dev/sda5
```

The file `/tmp/test-file` itself is at inode 181,288 on that device.

```
% ls --inode /tmp/test-file
 181288 /tmp/test-file
```

See Section 6.2, "Device Numbers," for more information about device numbers.

7.6 System Statistics

Two entries in `/proc` contain useful system statistics. The `/proc/loadavg` file contains information about the system load. The first three numbers represent the number of *active tasks* on the system—processes that are actually running—averaged over the last 1, 5, and 15 minutes. The next entry shows the instantaneous current number of *runnable tasks*—processes that are currently scheduled to run rather than being blocked in a system call—and the total number of processes on the system. The final entry is the process ID of the process that most recently ran.

The `/proc/uptime` file contains the length of time since the system was booted, as well as the amount of time since then that the system has been idle. Both are given as floating-point values, in seconds.

```
% cat /proc/uptime
3248936.18 3072330.49
```

The program in Listing 7.7 extracts the uptime and idle time from the system and displays them in friendly units.

Listing 7.7 **(*print-uptime.c*) Print the System Uptime and Idle Time**

```
#include <stdio.h>

/* Summarize a duration of time to standard output.  TIME is the
   amount of time, in seconds, and LABEL is a short descriptive label.  */

void print_time (char* label, long time)
{
```

continues

Listing 7.7 **Continued**

```
  /* Conversion constants.  */
  const long minute = 60;
  const long hour = minute * 60;
  const long day = hour * 24;
  /* Produce output.  */
  printf ("%s: %ld days, %ld:%02ld:%02ld\n", label, time / day,
          (time % day) / hour, (time % hour) / minute, time % minute);
}

int main ()
{
  FILE* fp;
  double uptime, idle_time;
  /* Read the system uptime and accumulated idle time from /proc/uptime.  */
  fp = fopen ("/proc/uptime", "r");
  fscanf (fp, "%lf %lf\n", &uptime, &idle_time);
  fclose (fp);
  /* Summarize it.  */
  print_time ("uptime   ", (long) uptime);
  print_time ("idle time", (long) idle_time);
  return 0;
}
```

The `uptime` command and the `sysinfo` system call (see Section 8.14, "`sysinfo`: Obtaining System Statistics") also can obtain the system's uptime. The `uptime` command also displays the load averages found in `/proc/loadavg`.

8

Linux System Calls

So far, we've presented a variety of functions that your program can invoke to perform system-related functions, such as parsing command-line options, manipulating processes, and mapping memory. If you look under the hood, you'll find that these functions fall into two categories, based on how they are implemented.

- A *library function* is an ordinary function that resides in a library external to your program. Most of the library functions we've presented so far are in the standard C library, `libc`. For example, `getopt_long` and `mkstemp` are functions provided in the C library.

  A call to a library function is just like any other function call. The arguments are placed in processor registers or onto the stack, and execution is transferred to the start of the function's code, which typically resides in a loaded shared library.

- A *system call* is implemented in the Linux kernel. When a program makes a system call, the arguments are packaged up and handed to the kernel, which takes over execution of the program until the call completes. A system call isn't an ordinary function call, and a special procedure is required to transfer control to the kernel. However, the GNU C library (the implementation of the standard C library provided with GNU/Linux systems) wraps Linux system calls with functions so that you can call them easily. Low-level I/O functions such as `open` and `read` are examples of system calls on Linux.

> The set of Linux system calls forms the most basic interface between programs and the Linux kernel. Each call presents a basic operation or capability.
>
> Some system calls are very powerful and can exert great influence on the system. For instance, some system calls enable you to shut down the Linux system or to allocate system resources and prevent other users from accessing them. These calls have the restriction that only processes running with superuser privilege (programs run by the root account) can invoke them. These calls fail if invoked by a nonsuperuser process.

Note that a library function may invoke one or more other library functions or system calls as part of its implementation.

Linux currently provides about 200 different system calls. A listing of system calls for your version of the Linux kernel is in `/usr/include/asm/unistd.h`. Some of these are for internal use by the system, and others are used only in implementing specialized library functions. In this chapter, we'll present a selection of system calls that are likely to be the most useful to application and system programmers.

Most of these system calls are declared in `<unistd.h>`.

8.1 Using *strace*

Before we start discussing system calls, it will be useful to present a command with which you can learn about and debug system calls. The `strace` command traces the execution of another program, listing any system calls the program makes and any signals it receives.

To watch the system calls and signals in a program, simply invoke `strace`, followed by the program and its command-line arguments. For example, to watch the system calls that are invoked by the `hostname`[1] command, use this command:

```
% strace hostname
```

This produces a couple screens of output. Each line corresponds to a single system call. For each call, the system call's name is listed, followed by its arguments (or abbreviated arguments, if they are very long) and its return value. Where possible, `strace` conveniently displays symbolic names instead of numerical values for arguments and return values, and it displays the fields of structures passed by a pointer into the system call. Note that `strace` does *not* show ordinary function calls.

In the output from `strace hostname`, the first line shows the `execve` system call that invokes the `hostname` program:[2]

```
execve("/bin/hostname", ["hostname"], [/* 49 vars */]) = 0
```

1. `hostname` invoked without any flags simply prints out the computer's hostname to standard output.

2. In Linux, the `exec` family of functions is implemented via the `execve` system call.

The first argument is the name of the program to run; the second is its argument list, consisting of only a single element; and the third is its environment list, which `strace` omits for brevity. The next 30 or so lines are part of the mechanism that loads the standard C library from a shared library file.

Toward the end are system calls that actually help do the program's work. The `uname` system call is used to obtain the system's hostname from the kernel,

```
uname({sys="Linux", node="myhostname", ...}) = 0
```

Observe that `strace` helpfully labels the fields (`sys` and `node`) of the structure argument. This structure is filled in by the system call—Linux sets the `sys` field to the operating system name and the `node` field to the system's hostname. The `uname` call is discussed further in Section 8.15, "`uname`."

Finally, the `write` system call produces output. Recall that file descriptor 1 corresponds to standard output. The third argument is the number of characters to write, and the return value is the number of characters that were actually written.

```
write(1, "myhostname\n", 11)            = 11
```

This may appear garbled when you run `strace` because the output from the `hostname` program itself is mixed in with the output from `strace`.

If the program you're tracing produces lots of output, it is sometimes more convenient to redirect the output from `strace` into a file. Use the option `-o` *`filename`* to do this.

Understanding all the output from `strace` requires detailed familiarity with the design of the Linux kernel and execution environment. Much of this is of limited interest to application programmers. However, some understanding is useful for debugging tricky problems or understanding how other programs work.

8.2 *access*: Testing File Permissions

The `access` system call determines whether the calling process has access permission to a file. It can check any combination of read, write, and execute permission, and it can also check for a file's existence.

The `access` call takes two arguments. The first is the path to the file to check. The second is a bitwise or of `R_OK`, `W_OK`, and `X_OK`, corresponding to read, write, and execute permission. The return value is 0 if the process has all the specified permissions. If the file exists but the calling process does not have the specified permissions, `access` returns –1 and sets `errno` to `EACCES` (or `EROFS`, if write permission was requested for a file on a read-only file system).

If the second argument is `F_OK`, `access` simply checks for the file's existence. If the file exists, the return value is 0; if not, the return value is –1 and `errno` is set to `ENOENT`. Note that `errno` may instead be set to `EACCES` if a directory in the file path is inaccessible.

The program shown in Listing 8.1 uses `access` to check for a file's existence and to determine read and write permissions. Specify the name of the file to check on the command line.

Listing 8.1 **(*check-access.c*) Check File Access Permissions**

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char* path = argv[1];
  int rval;

  /* Check file existence.  */
  rval = access (path, F_OK);
  if (rval == 0)
    printf ("%s exists\n", path);
  else {
    if (errno == ENOENT)
      printf ("%s does not exist\n", path);
    else if (errno == EACCES)
      printf ("%s is not accessible\n", path);
    return 0;
  }

  /* Check read access.  */
  rval = access (path, R_OK);
  if (rval == 0)
    printf ("%s is readable\n", path);
  else
    printf ("%s is not readable (access denied)\n", path);

  /* Check write access.  */
  rval = access (path, W_OK);
  if (rval == 0)
    printf ("%s is writable\n", path);
  else if (errno == EACCES)
    printf ("%s is not writable (access denied)\n", path);
  else if (errno == EROFS)
    printf ("%s is not writable (read-only filesystem)\n", path);
  return 0;
}
```

For example, to check access permissions for a file named `README` on a CD-ROM, invoke it like this:

```
% ./check-access /mnt/cdrom/README
/mnt/cdrom/README exists
/mnt/cdrom/README is readable
/mnt/cdrom/README is not writable (read-only filesystem)
```

8.3 *fcntl*: Locks and Other File Operations

The `fcntl` system call is the access point for several advanced operations on file descriptors. The first argument to `fcntl` is an open file descriptor, and the second is a value that indicates which operation is to be performed. For some operations, `fcntl` takes an additional argument. We'll describe here one of the most useful `fcntl` operations, file locking. See the `fcntl` man page for information about the others.

The `fcntl` system call allows a program to place a read lock or a write lock on a file, somewhat analogous to the mutex locks discussed in Chapter 5, "Interprocess Communication." A read lock is placed on a readable file descriptor, and a write lock is placed on a writable file descriptor. More than one process may hold a read lock on the same file at the same time, but only one process may hold a write lock, and the same file may not be both locked for read and locked for write. Note that placing a lock does not actually prevent other processes from opening the file, reading from it, or writing to it, unless they acquire locks with `fcntl` as well.

To place a lock on a file, first create and zero out a `struct flock` variable. Set the `l_type` field of the structure to `F_RDLCK` for a read lock or `F_WRLCK` for a write lock. Then call `fcntl`, passing a file descriptor to the file, the `F_SETLCKW` operation code, and a pointer to the `struct flock` variable. If another process holds a lock that prevents a new lock from being acquired, `fcntl` blocks until that lock is released.

The program in Listing 8.2 opens a file for writing whose name is provided on the command line, and then places a write lock on it. The program waits for the user to hit Enter and then unlocks and closes the file.

Listing 8.2 **(*lock-file.c*) Create a Write Lock with *fcntl***

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char* file = argv[1];
  int fd;
  struct flock lock;

  printf ("opening %s\n", file);
  /* Open a file descriptor to the file.  */
  fd = open (file, O_WRONLY);
  printf ("locking\n");
  /* Initialize the flock structure.  */
  memset (&lock, 0, sizeof(lock));
  lock.l_type = F_WRLCK;
  /* Place a write lock on the file.  */
  fcntl (fd, F_SETLKW, &lock);
```

continues

Listing 8.2 **Continued**

```
  printf ("locked; hit Enter to unlock... ");
  /* Wait for the user to hit Enter.  */
  getchar ();

  printf ("unlocking\n");
  /* Release the lock.  */
  lock.l_type = F_UNLCK;
  fcntl (fd, F_SETLKW, &lock);

  close (fd);
  return 0;
}
```

Compile and run the program on a test file—say, `/tmp/test-file`—like this:

```
% cc -o lock-file lock-file.c
% touch /tmp/test-file
% ./lock-file /tmp/test-file
opening /tmp/test-file
locking
locked; hit Enter to unlock...
```

Now, in another window, try running it again on the same file.

```
% ./lock-file /tmp/test-file
opening /tmp/test-file
locking
```

Note that the second instance is blocked while attempting to lock the file. Go back to the first window and press Enter:

```
unlocking
```

The program running in the second window immediately acquires the lock.

If you prefer `fcntl` not to block if the call cannot get the lock you requested, use `F_SETLK` instead of `F_SETLKW`. If the lock cannot be acquired, `fcntl` returns –1 immediately.

Linux provides another implementation of file locking with the `flock` call. The `fcntl` version has a major advantage: It works with files on NFS[3] file systems (as long as the NFS server is reasonably recent and correctly configured). So, if you have access to two machines that both mount the same file system via NFS, you can repeat the previous example using two different machines. Run `lock-file` on one machine, specifying a file on an NFS file system, and then run it again on another machine, specifying the same file. NFS wakes up the second program when the lock is released by the first program.

3. *Network File System* (NFS) is a common network file sharing technology, comparable to Windows' shares and network drives.

8.4 *fsync* and *fdatasync*: Flushing Disk Buffers

On most operating systems, when you write to a file, the data is not immediately written to disk. Instead, the operating system caches the written data in a memory buffer, to reduce the number of required disk writes and improve program responsiveness. When the buffer fills or some other condition occurs (for instance, enough time elapses), the system writes the cached data to disk all at one time.

Linux provides caching of this type as well. Normally, this is a great boon to performance. However, this behavior can make programs that depend on the integrity of disk-based records unreliable. If the system goes down suddenly—for instance, due to a kernel crash or power outage—any data written by a program that is in the memory cache but has not yet been written to disk is lost.

For example, suppose that you are writing a transaction-processing program that keeps a journal file. The journal file contains records of all transactions that have been processed so that if a system failure occurs, the state of the transaction data can be reconstructed. It is obviously important to preserve the integrity of the journal file—whenever a transaction is processed, its journal entry should be sent to the disk drive immediately.

To help you implement this, Linux provides the `fsync` system call. It takes one argument, a writable file descriptor, and flushes to disk any data written to this file. The `fsync` call doesn't return until the data has physically been written.

The function in Listing 8.3 illustrates the use of `fsync`. It writes a single-line entry to a journal file.

Listing 8.3 **(*write_journal_entry.c*) Write and Sync a Journal Entry**

```
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

const char* journal_filename = "journal.log";

void write_journal_entry (char* entry)
{
  int fd = open (journal_filename, O_WRONLY | O_CREAT | O_APPEND, 0660);
  write (fd, entry, strlen (entry));
  write (fd, "\n", 1);
  fsync (fd);
  close (fd);
}
```

Another system call, `fdatasync` does the same thing. However, although `fsync` guarantees that the file's modification time will be updated, `fdatasync` does not; it guarantees only that the file's data will be written. This means that in principal, `fdatasync` can execute faster than `fsync` because it needs to force only one disk write instead of two.

However, in current versions of Linux, these two system calls actually do the same thing, both updating the file's modification time.

The `fsync` system call enables you to force a buffer write explicitly. You can also open a file for *synchronous I/O*, which causes all writes to be committed to disk immediately. To do this, specify the `O_SYNC` flag when opening the file with the `open` call.

8.5 *getrlimit* and *setrlimit*: Resource Limits

The `getrlimit` and `setrlimit` system calls allow a process to read and set limits on the system resources that it can consume. You may be familiar with the `ulimit` shell command, which enables you to restrict the resource usage of programs you run;[4] these system calls allow a program to do this programmatically.

For each resource there are two limits, the *hard limit* and the *soft limit*. The soft limit may never exceed the hard limit, and only processes with superuser privilege may change the hard limit. Typically, an application program will reduce the soft limit to place a throttle on the resources it uses.

Both `getrlimit` and `setrlimit` take as arguments a code specifying the resource limit type and a pointer to a `structrlimit` variable. The `getrlimit` call fills the fields of this structure, while the `setrlimit` call changes the limit based on its contents. The `rlimit` structure has two fields: `rlim_cur` is the soft limit, and `rlim_max` is the hard limit.

Some of the most useful resource limits that may be changed are listed here, with their codes:

- `RLIMIT_CPU`—The maximum CPU time, in seconds, used by a program. This is the amount of time that the program is actually executing on the CPU, which is not necessarily the same as wall-clock time. If the program exceeds this time limit, it is terminated with a `SIGXCPU` signal.
- `RLIMIT_DATA`—The maximum amount of memory that a program can allocate for its data. Additional allocation beyond this limit will fail.
- `RLIMIT_NPROC`—The maximum number of child processes that can be running for this user. If the process calls `fork` and too many processes belonging to this user are running on the system, `fork` fails.
- `RLIMIT_NOFILE`—The maximum number of file descriptors that the process may have open at one time.

See the `setrlimit` man page for a full list of system resources.

The program in Listing 8.4 illustrates setting the limit on CPU time consumed by a program. It sets a 1-second CPU time limit and then spins in an infinite loop. Linux kills the process soon afterward, when it exceeds 1 second of CPU time.

4. See the man page for your shell for more information about `ulimit`.

Listing 8.4 (*limit-cpu.c*) CPU Time Limit Demonstration

```
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

int main ()
{
  struct rlimit rl;

  /* Obtain the current limits.  */
  getrlimit (RLIMIT_CPU, &rl);
  /* Set a CPU limit of 1 second.  */
  rl.rlim_cur = 1;
  setrlimit (RLIMIT_CPU, &rl);
  /* Do busy work.  */
  while (1);

  return 0;
}
```

When the program is terminated by `SIGXCPU`, the shell helpfully prints out a message interpreting the signal:

```
% ./limit_cpu
CPU time limit exceeded
```

8.6 *getrusage*: Process Statistics

The `getrusage` system call retrieves process statistics from the kernel. It can be used to obtain statistics either for the current process by passing `RUSAGE_SELF` as the first argument, or for all terminated child processes that were forked by this process and its children by passing `RUSAGE_CHILDREN`. The second argument to `rusage` is a pointer to a `struct rusage` variable, which is filled with the statistics.

A few of the more interesting fields in `struct rusage` are listed here:

- `ru_utime`—A `struct timeval` field containing the amount of *user time*, in seconds, that the process has used. User time is CPU time spent executing the user program, rather than in kernel system calls.
- `ru_stime`—A `struct timeval` field containing the amount of *system time*, in seconds, that the process has used. System time is the CPU time spent executing system calls on behalf of the process.
- `ru_maxrss`—The largest amount of physical memory occupied by the process's data at one time over the course of its execution.

The `getrusage` man page lists all the available fields. See Section 8.7, "`gettimeofday`: Wall-Clock Time," for information about `struct timeval`.

The function in Listing 8.5 prints out the current process's user and system time.

Listing 8.5 **(*print-cpu-times.c*) Display Process User and System Times**

```
#include <stdio.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

void print_cpu_time()
{
  struct rusage usage;
  getrusage (RUSAGE_SELF, &usage);
  printf ("CPU time: %ld.%06ld sec user, %ld.%06ld sec system\n",
          usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
          usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
}
```

8.7 *gettimeofday*: Wall-Clock Time

The `gettimeofday` system call gets the system's wall-clock time. It takes a pointer to a `struct timeval` variable. This structure represents a time, in seconds, split into two fields. The `tv_sec` field contains the integral number of seconds, and the `tv_usec` field contains an additional number of microseconds. This `struct timeval` value represents the number of seconds elapsed since the start of the *UNIX epoch*, on midnight UTC on January 1, 1970. The `gettimeofday` call also takes a second argument, which should be `NULL`. Include `<sys/time.h>` if you use this system call.

The number of seconds in the UNIX epoch isn't usually a very handy way of representing dates. The `localtime` and `strftime` library functions help manipulate the return value of `gettimeofday`. The `localtime` function takes a pointer to the number of seconds (the `tv_sec` field of `struct timeval`) and returns a pointer to a `struct tm` object. This structure contains more useful fields, which are filled according to the local time zone:

- `tm_hour`, `tm_min`, `tm_sec`—The time of day, in hours, minutes, and seconds.
- `tm_year`, `tm_mon`, `tm_day`—The year, month, and date.
- `tm_wday`—The day of the week. Zero represents Sunday.
- `tm_yday`—The day of the year.
- `tm_isdst`—A flag indicating whether daylight savings time is in effect.

The `strftime` function additionally can produce from the `struct tm` pointer a customized, formatted string displaying the date and time. The format is specified in a manner similar to `printf`, as a string with embedded codes indicating which time fields to include. For example, this format string

```
"%Y-%m-%d %H:%M:%S"
```

specifies the date and time in this form:

```
2001-01-14 13:09:42
```

Pass `strftime` a character buffer to receive the string, the length of that buffer, the format string, and a pointer to a `struct tm` variable. See the `strftime` man page for a complete list of codes that can be used in the format string. Notice that neither `localtime` nor `strftime` handles the fractional part of the current time more precise than 1 second (the `tv_usec` field of `struct timeval`). If you want this in your formatted time strings, you'll have to include it yourself.

Include `<time.h>` if you call `localtime` or `strftime`.

The function in Listing 8.6 prints the current date and time of day, down to the millisecond.

Listing 8.6 **(*print-time.c*) Print Date and Time**

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

void print_time ()
{
  struct timeval tv;
  struct tm* ptm;
  char time_string[40];
  long milliseconds;

  /* Obtain the time of day, and convert it to a tm struct.  */
  gettimeofday (&tv, NULL);
  ptm = localtime (&tv.tv_sec);
  /* Format the date and time, down to a single second.  */
  strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);
  /* Compute milliseconds from microseconds.  */
  milliseconds = tv.tv_usec / 1000;
  /* Print the formatted time, in seconds, followed by a decimal point
     and the milliseconds.  */
  printf ("%s.%03ld\n", time_string, milliseconds);
}
```

8.8 The *mlock* Family: Locking Physical Memory

The `mlock` family of system calls allows a program to lock some or all of its address space into physical memory. This prevents Linux from paging this memory to swap space, even if the program hasn't accessed it for a while.

A time-critical program might lock physical memory because the time delay of paging memory out and back may be too long or too unpredictable. High-security applications may also want to prevent sensitive data from being written out to a swap file, where they might be recovered by an intruder after the program terminates.

Locking a region of memory is as simple as calling `mlock` with a pointer to the start of the region and the region's length. Linux divides memory into *pages* and can lock only entire pages at a time; each page that contains part of the memory region specified to `mlock` is locked. The `getpagesize` function returns the system's page size, which is 4KB on x86 Linux.

For example, to allocate 32MB of address space and lock it into RAM, you would use this code:

```
const int alloc_size = 32 * 1024 * 1024;
char* memory = malloc (alloc_size);
mlock (memory, alloc_size);
```

Note that simply allocating a page of memory and locking it with `mlock` doesn't reserve physical memory for the calling process because the pages may be copy-on-write.[5] Therefore, you should write a dummy value to each page as well:

```
size_t i;
size_t page_size = getpagesize ();
for (i = 0; i < alloc_size; i += page_size)
  memory[i] = 0;
```

The write to each page forces Linux to assign a unique, unshared memory page to the process for that page.

To unlock a region, call `munlock`, which takes the same arguments as `mlock`.

If you want your program's entire address space locked into physical memory, call `mlockall`. This system call takes a single flag argument: `MCL_CURRENT` locks all currently allocated memory, but future allocations are not locked; `MCL_FUTURE` locks all pages that are allocated after the call. Use `MCL_CURRENT|MCL_FUTURE` to lock into memory both current and subsequent allocations.

Locking large amounts of memory, especially using `mlockall`, can be dangerous to the entire Linux system. Indiscriminate memory locking is a good method of bringing your system to a grinding halt because other running processes are forced to compete for smaller physical memory resources and swap rapidly into and back out of memory (this is known as *thrashing*). If you lock too much memory, the system will run out of memory entirely and Linux will start killing off processes.

For this reason, only processes with superuser privilege may lock memory with `mlock` or `mlockall`. If a nonsuperuser process calls one of these functions, it will fail, return –1, and set `errno` to `EPERM`.

The `munlockall` call unlocks all memory locked by the current process, including memory locked with `mlock` and `mlockall`.

5. *Copy-on-write* means that Linux makes a private copy of a page of memory for a process only when that process writes a value somewhere into it.

A convenient way to monitor the memory usage of your program is to use the `top` command. In the output from `top`, the `SIZE` column displays the virtual address space size of each program (the total size of your program's code, data, and stack, some of which may be paged out to swap space). The `RSS` column (for *resident set size*) shows the size of physical memory that each program currently resides in. The sum of all the RSS values for all running programs cannot exceed your computer's physical memory size, and the sum of all address space sizes is limited to 2GB (for 32-bit versions of Linux).

Include `<sys/mman.h>` if you use any of the `mlock` system calls.

8.9 *mprotect*: Setting Memory Permissions

In Section 5.3, "Mapped Memory," we showed how to use the `mmap` system call to map a file into memory. Recall that the third argument to `mmap` is a bitwise or of memory protection flags `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC` for read, write, and execute permission, respectively, or `PROT_NONE` for no memory access. If a program attempts to perform an operation on a memory location that is not allowed by these permissions, it is terminated with a `SIGSEGV` (segmentation violation) signal.

After memory has been mapped, these permissions can be modified with the `mprotect` system call. The arguments to `mprotect` are an address of a memory region, the size of the region, and a set of protection flags. The memory region must consist of entire pages: The address of the region must be aligned to the system's page size, and the length of the region must be a page size multiple. The protection flags for these pages are replaced with the specified value.

> **Obtaining Page-Aligned Memory**
>
> Note that memory regions returned by `malloc` are typically not page-aligned, even if the size of the memory is a multiple of the page size. If you want to protect memory obtained from `malloc`, you will have to allocate a larger memory region and find a page-aligned region within it.
>
> Alternately, you can use the `mmap` system call to bypass `malloc` and allocate page-aligned memory directly from the Linux kernel. See Section 5.3, "Mapped Memory," for details.

For example, suppose that your program allocates a page of memory by mapping `/dev/zero`, as described in Section 5.3.5, "Other Uses for `mmap`." The memory is initially both readable and writable.

```
int fd = open ("/dev/zero", O_RDONLY);
char* memory = mmap (NULL, page_size, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE, fd, 0);
close (fd);
```

Later, your program could make the memory read-only by calling `mprotect`:

```
mprotect (memory, page_size, PROT_READ);
```

An advanced technique to monitor memory access is to protect the region of memory using `mmap` or `mprotect` and then handle the `SIGSEGV` signal that Linux sends to the program when it tries to access that memory. The example in Listing 8.7 illustrates this technique.

Listing 8.7 **(*mprotect.c*) Detect Memory Access Using *mprotect***

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

static int alloc_size;
static char* memory;

void segv_handler (int signal_number)
{
  printf ("memory accessed!\n");
  mprotect (memory, alloc_size, PROT_READ | PROT_WRITE);
}

int main ()
{
  int fd;
  struct sigaction sa;

  /* Install segv_handler as the handler for SIGSEGV.  */
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &segv_handler;
  sigaction (SIGSEGV, &sa, NULL);

  /* Allocate one page of memory by mapping /dev/zero.  Map the memory
     as write-only, initially.  */
  alloc_size = getpagesize ();
  fd = open ("/dev/zero", O_RDONLY);
  memory = mmap (NULL, alloc_size, PROT_WRITE, MAP_PRIVATE, fd, 0);
  close (fd);
  /* Write to the page to obtain a private copy.  */
  memory[0] = 0;
  /* Make the memory unwritable.  */
  mprotect (memory, alloc_size, PROT_NONE);

  /* Write to the allocated memory region.  */
  memory[0] = 1;
```

```
  /* All done; unmap the memory.  */
  printf ("all done\n");
  munmap (memory, alloc_size);
  return 0;
}
```

The program follows these steps:

1. The program installs a signal handler for `SIGSEGV`.
2. The program allocates a page of memory by mapping `/dev/zero` and writing a value to the allocated page to obtain a private copy.
3. The program protects the memory by calling `mprotect` with the `PROT_NONE` permission.
4. When the program subsequently writes to memory, Linux sends it `SIGSEGV`, which is handled by `segv_handler`. The signal handler unprotects the memory, which allows the memory access to proceed.
5. When the signal handler completes, control returns to `main`, where the program deallocates the memory using `munmap`.

8.10 *nanosleep*: High-Precision Sleeping

The `nanosleep` system call is a high-precision version of the standard UNIX `sleep` call. Instead of sleeping an integral number of seconds, `nanosleep` takes as its argument a pointer to a `struct timespec` object, which can express time to nanosecond precision. However, because of the details of how the Linux kernel works, the actual precision provided by `nanosleep` is 10 milliseconds—still better than that afforded by `sleep`. This additional precision can be useful, for instance, to schedule frequent operations with short time intervals between them.

The `struct timespec` structure has two fields: `tv_sec`, the integral number of seconds, and `tv_nsec`, an additional number of milliseconds. The value of `tv_nsec` must be less than 10^9.

The `nanosleep` call provides another advantage over `sleep`. As with `sleep`, the delivery of a signal interrupts the execution of `nanosleep`, which sets `errno` to `EINTR` and returns –1. However, `nanosleep` takes a second argument, another pointer to a `struct timespec` object, which, if not null, is filled with the amount of time remaining (that is, the difference between the requested sleep time and the actual sleep time). This makes it easy to resume the sleep operation.

The function in Listing 8.8 provides an alternate implementation of `sleep`. Unlike the ordinary system call, this function takes a floating-point value for the number of seconds to sleep and restarts the sleep operation if it's interrupted by a signal.

Listing 8.8 **(*better_sleep.c*) High-Precision Sleep Function**

```
#include <errno.h>
#include <time.h>

int better_sleep (double sleep_time)
{
  struct timespec tv;
  /* Construct the timespec from the number of whole seconds...  */
  tv.tv_sec = (time_t) sleep_time;
  /* ... and the remainder in nanoseconds.  */
  tv.tv_nsec = (long) ((sleep_time - tv.tv_sec) * 1e+9);

  while (1)
  {
    /* Sleep for the time specified in tv.  If interrupted by a
       signal, place the remaining time left to sleep back into tv.  */
    int rval = nanosleep (&tv, &tv);
    if (rval == 0)
      /* Completed the entire sleep time; all done.  */
      return 0;
    else if (errno == EINTR)
      /* Interrupted by a signal.  Try again.  */
      continue;
    else
      /* Some other error; bail out.  */
      return rval;
  }
  return 0;
}
```

8.11 *readlink*: Reading Symbolic Links

The `readlink` system call retrieves the target of a symbolic link. It takes three arguments: the path to the symbolic link, a buffer to receive the target of the link, and the length of that buffer. Unusually, `readlink` does not NUL-terminate the target path that it fills into the buffer. It does, however, return the number of characters in the target path, so NUL-terminating the string is simple.

If the first argument to `readlink` points to a file that isn't a symbolic link, `readlink` sets `errno` to `EINVAL` and returns –1.

The small program in Listing 8.9 prints the target of the symbolic link specified on its command line.

Listing 8.9 **(*print-symlink.c*) Print the Target of a Symbolic Link**

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char target_path[256];
  char* link_path = argv[1];

  /* Attempt to read the target of the symbolic link.  */
  int len = readlink (link_path, target_path, sizeof (target_path));

  if (len == -1) {
    /* The call failed.  */
    if (errno == EINVAL)
      /* It's not a symbolic link; report that.  */
      fprintf (stderr, "%s is not a symbolic link\n", link_path);
    else
      /* Some other problem occurred; print the generic message.  */
      perror ("readlink");
    return 1;
  }
  else {
    /* NUL-terminate the target path.  */
    target_path[len] = '\0';
    /* Print it.  */
    printf ("%s\n", target_path);
    return 0;
  }
}
```

For example, here's how you could make a symbolic link and use `print-symlink` to read it back:

```
% ln -s /usr/bin/wc my_link
% ./print-symlink my_link
/usr/bin/wc
```

8.12 *sendfile*: Fast Data Transfers

The `sendfile` system call provides an efficient mechanism for copying data from one file descriptor to another. The file descriptors may be open to disk files, sockets, or other devices.

Typically, to copy from one file descriptor to another, a program allocates a fixed-size buffer, copies some data from one descriptor into the buffer, writes the buffer out to the other descriptor, and repeats until all the data has been copied. This is inefficient in both time and space because it requires additional memory for the buffer and performs an extra copy of the data into that buffer.

Using `sendfile`, the intermediate buffer can be eliminated. Call `sendfile`, passing the file descriptor to write to; the descriptor to read from; a pointer to an offset variable; and the number of bytes to transfer. The offset variable contains the offset in the input file from which the read should start (0 indicates the beginning of the file) and is updated to the position in the file after the transfer. The return value is the number of bytes transferred. Include `<sys/sendfile.h>` in your program if it uses `sendfile`.

The program in Listing 8.10 is a simple but extremely efficient implementation of a file copy. When invoked with two filenames on the command line, it copies the contents of the first file into a file named by the second. It uses `fstat` to determine the size, in bytes, of the source file.

Listing 8.10 **(*copy.c*) File Copy Using *sendfile***

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  int read_fd;
  int write_fd;
  struct stat stat_buf;
  off_t offset = 0;

  /* Open the input file.  */
  read_fd = open (argv[1], O_RDONLY);
  /* Stat the input file to obtain its size.  */
  fstat (read_fd, &stat_buf);
  /* Open the output file for writing, with the same permissions as the
     source file.  */
  write_fd = open (argv[2], O_WRONLY | O_CREAT, stat_buf.st_mode);
  /* Blast the bytes from one file to the other.  */
  sendfile (write_fd, read_fd, &offset, stat_buf.st_size);
  /* Close up.  */
  close (read_fd);
  close (write_fd);

  return 0;
}
```

The `sendfile` call can be used in many places to make copies more efficient. One good example is in a Web server or other network daemon, that serves the contents of a file over the network to a client program. Typically, a request is received from a socket connected to the client computer. The server program opens a local disk file to

retrieve the data to serve and writes the file's contents to the network socket. Using `sendfile` can speed up this operation considerably. Other steps need to be taken to make the network transfer as efficient as possible, such as setting the socket parameters correctly. However, these are outside the scope of this book.

8.13 *setitimer*: Setting Interval Timers

The `setitimer` system call is a generalization of the `alarm` call. It schedules the delivery of a signal at some point in the future after a fixed amount of time has elapsed.

A program can set three different types of timers with `setitimer`:

- If the timer code is `ITIMER_REAL`, the process is sent a `SIGALRM` signal after the specified wall-clock time has elapsed.
- If the timer code is `ITIMER_VIRTUAL`, the process is sent a `SIGVTALRM` signal after the process has executed for the specified time. Time in which the process is not executing (that is, when the kernel or another process is running) is not counted.
- If the timer code is `ITIMER_PROF`, the process is sent a `SIGPROF` signal when the specified time has elapsed either during the process's own execution or the execution of a system call on behalf of the process.

The first argument to `setitimer` is the timer code, specifying which timer to set. The second argument is a pointer to a `struct itimerval` object specifying the new settings for that timer. The third argument, if not null, is a pointer to another `struct itimerval` object that receives the old timer settings.

A `struct itimerval` variable has two fields:

- `it_value` is a `struct timeval` field that contains the time until the timer next expires and a signal is sent. If this is 0, the timer is disabled.
- `it_interval` is another `struct timeval` field containing the value to which the timer will be reset after it expires. If this is 0, the timer will be disabled after it expires. If this is nonzero, the timer is set to expire repeatedly after this interval.

The `struct timeval` type is described in Section 8.7, "`gettimeofday`: Wall-Clock Time."

The program in Listing 8.11 illustrates the use of `setitimer` to track the execution time of a program. A timer is configured to expire every 250 milliseconds and send a `SIGVTALRM` signal.

Listing 8.11 **(*itemer.c*) Timer Example**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
```

continues

Listing 8.11 **Continued**

```
void timer_handler (int signum)
{
  static int count = 0;
  printf ("timer expired %d times\n", ++count);
}

int main ()
{
  struct sigaction sa;
  struct itimerval timer;

  /* Install timer_handler as the signal handler for SIGVTALRM.  */
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &timer_handler;
  sigaction (SIGVTALRM, &sa, NULL);

  /* Configure the timer to expire after 250 msec...  */
  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 250000;
  /* ... and every 250 msec after that.  */
  timer.it_interval.tv_sec = 0;
  timer.it_interval.tv_usec = 250000;
  /* Start a virtual timer.  It counts down whenever this process is
     executing.  */
  setitimer (ITIMER_VIRTUAL, &timer, NULL);

  /* Do busy work.  */
  while (1);
}
```

8.14 *sysinfo*: Obtaining System Statistics

The `sysinfo` system call fills a structure with system statistics. Its only argument is a pointer to a `struct sysinfo`. Some of the more interesting fields of `struct sysinfo` that are filled include these:

- `uptime`—Time elapsed since the system booted, in seconds
- `totalram`—Total available physical RAM
- `freeram`—Free physical RAM
- `procs`—Number of processes on the system

See the `sysinfo` man page for a full description of `structsysinfo`. Include `<linux/kernel.h>`, `<linux/sys.h>`, and `<sys/sysinfo.h>` if you use `sysinfo`.

The program in Listing 8.12 prints some statistics about the current system.

Listing 8.12 **(*sysinfo.c*) Print System Statistics**

```
#include <linux/kernel.h>
#include <linux/sys.h>
#include <stdio.h>
#include <sys/sysinfo.h>

int main ()
{
  /* Conversion constants.  */
  const long minute = 60;
  const long hour = minute * 60;
  const long day = hour * 24;
  const double megabyte = 1024 * 1024;
  /* Obtain system statistics.  */
  struct sysinfo si;
  sysinfo (&si);
  /* Summarize interesting values.  */
  printf ("system uptime : %ld days, %ld:%02ld:%02ld\n",
          si.uptime / day, (si.uptime % day) / hour,
          (si.uptime % hour) / minute, si.uptime % minute);
  printf ("total RAM     : %5.1f MB\n", si.totalram / megabyte);
  printf ("free RAM      : %5.1f MB\n", si.freeram / megabyte);
  printf ("process count : %d\n", si.procs);
  return 0;
}
```

8.15 *uname*

The `uname` system call fills a structure with various system information, including the computer's network name and domain name, and the operating system version it's running. Pass `uname` a single argument, a pointer to a `struct utsname` object. Include `<sys/utsname.h>` if you use `uname`.

The call to `uname` fills in these fields:

- `sysname`—The name of the operating system (such as Linux).
- `release`, `version`—The Linux kernel release number and version level.
- `machine`—Some information about the hardware platform running Linux. For x86 Linux, this is i386 or i686, depending on the processor.
- `node`—The computer's unqualified hostname.
- `__domain`—The computer's domain name.

Each of these fields is a character string.

The small program in Listing 8.13 prints the Linux release and version number and the hardware information.

Listing 8.13 **(*print-uname*) Print Linux Version Number and Hardware Information**

```
#include <stdio.h>
#include <sys/utsname.h>

int main ()
{
  struct utsname u;
  uname (&u);
  printf ("%s release %s (version %s) on %s\n", u.sysname, u.release,
          u.version, u.machine);
  return 0;
}
```