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`.