

## Linux System Calls

**S**<sub>O FAR</sub>, 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<sup>1</sup> 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:<sup>2</sup>

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

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<sup>3</sup> 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, 0_WRONLY | 0_CREAT | 0_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 **0\_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;<sup>4</sup> 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

#### 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.<sup>5</sup> 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.

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", 0 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.
- 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<sup>9</sup>.

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 fixedsize 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], 0 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], 0_WRONLY | 0_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>
```

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/svs.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);
                    : %5.1f MB\n", si.freeram / megabyte);
 printf ("free RAM
 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;
}
```