14 GNU/Linux Process Model

In This Chapter

- Creating Processes with `fork()`
- Review of Process-Related API Functions
- Raising and Catching Signals
- Available Signals and Their Uses
- GNU/Linux Process-Related Commands

INTRODUCTION

This chapter introduces the GNU/Linux process model. It defines elements of a process, how processes communicate with each other, and how to control and monitor them. First, the chapter addresses a quick review of fundamental APIs and then follows up with a more detailed review, complete with sample applications that illustrate each technique.

GNU/LINUX PROCESSES

GNU/Linux presents two fundamental types of processes. These are *kernel threads* and *user processes*. The focus here is on user processes (those created by `fork` and `clone`). Kernel threads are created within the kernel context via the `kernel_thread()` function.

When a subprocess is created (via `fork`), a new child task is created with a copy of the memory used by the original parent task. This memory is separate between the two processes. Any variables present when the `fork` takes place are available to the child. But after the `fork` completes, any changes that the parent makes to a variable are not seen by the child. This is important to consider when using the `fork` API function.

NOTE

When a new task is created, the memory space used by the parent isn't actually copied to the child. Instead, both the parent and child reference the same memory space, with the memory pages marked as copy-on-write. When any of the processes attempt to write to the memory, a new set of memory pages is created for the process that is private to it alone. In this way, creating a new process is an efficient mechanism, with copying of the memory space deferred until writes take place. In the default case, the child process inherits open file descriptors, the memory image, and CPU state (such as the PC and assorted registers).

Certain elements are not copied from the parent and instead are created specifically for the child. The following sections take a look at examples of these. What's important to understand at this stage is that a process can create subprocesses (known as *children*) and generally control them.

WHIRLWIND TOUR OF PROCESS APIs

As defined previously, you can create a new process with the `fork` or `clone` API function. But in fact, you create a new process every time you execute a command or start a program. Consider the simple program shown in Listing 14.1.

LISTING 14.1 First Process Example (on the CD-ROM at `./source/ch14/process.c`)

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

int main()
{
  pid_t myPid;
  pid_t myParentPid;
  gid_t myGid;
  uid_t myUid;

  myPid = getpid();
```

```
    myParentPid = getppid();
    myGid = getgid();
    myUid = getuid();

    printf( "my process id is %d\n", myPid );

    printf( "my parent's process id is %d\n", myParentPid );

    printf( "my group id is %d\n", myGid );

    printf( "my user id is %d\n", myUid );

    return 0;
}
```

Every process in GNU/Linux has a unique identifier called a process ID (or `pid`). Every process also has a parent (except for the `init` process). In Listing 14.1, you use the `getpid()` function to get the current process ID and the `getppid()` function to retrieve the process's parent ID. Then you grab the group ID and the user ID using `getuid()` and `getgid()`.

If you were to compile and then execute this application, you would see the following:

```
$ ./process
my process id is 10932
my parent's process id is 10795
my group id is 500
my user id is 500
$
```

You see the process ID is 10932, and the parent is 10795 (our bash shell). If you execute the application again, you see the following:

```
$ ./process
my process id is 10933
my parent's process id is 10795
my group id is 500
my user id is 500
$
```

Note that your process ID has changed, but all other values have remained the same. This is expected, because the only thing you've done is create a new process that performs its I/O and then exits. Each time a new process is created, a new process ID is allocated to it.

CREATING A SUBPROCESS WITH `fork`

Now it's time to move on to the real topic of this chapter, creating new processes within a given process. The `fork` API function is the most common method to achieve this.

The `fork` call is an oddity when you consider what is actually occurring. When the `fork` API function returns, the split occurs, and the return value from `fork` identifies in which context the process is running. Consider the following code snippet:

```
pid_t pid;
...
pid = fork();
if (pid > 0) {
  /* Parent context, child is pid */
} else if (pid == 0) {
  /* Child context */
} else {
  /* Parent context, error occurred, no child created */
}
```

You see here three possibilities from the return of the `fork` call. When the return value of `fork` is greater than zero, then you're in the parent context and the value represents the process ID of the child. When the return value is zero, then you're in the child process's context. Finally, any other value (less than zero) represents an error and is performed within the context of the parent.

Now it's time to look at a sample application of `fork` (shown in Listing 14.2). This working example illustrates the `fork` call, identifying the contexts. At line 11, you call `fork` to split your process into parent and child. Both the parent and child emit some text to standard-out so you can see each execution. Note that a shared variable (`role`) is updated by both parent and child and emitted at line 45.

LISTING 14.2 Working Example of the `fork` Call (on the CD-ROM at `./source/ch14/smplfork.c`)

```
1:        #include <sys/types.h>
2:        #include <unistd.h>
3:        #include <errno.h>
4:
5:        int main()
6:        {
7:          pid_t ret;
8:          int   status, i;
```

```
  int   role = -1;

  ret = fork();

  if (ret > 0) {

    printf(“Parent: This is the parent process (pid %d)\n”,
             getpid());

    for (i = 0 ; i < 10 ; i++) {
      printf(“Parent: At count %d\n”, i);
      sleep(1);
    }

    ret = wait( &status );

    role = 0;

  } else if (ret == 0) {

    printf(“Child: This is the child process (pid %d)\n”,
             getpid());

    for (i = 0 ; i < 10 ; i++) {
      printf(“Child: At count %d\n”, i);
      sleep(1);
    }

    role = 1;

  } else {

    printf(“Parent: Error trying to fork() (%d)\n”, errno);

  }

  printf(“%s: Exiting...\n”,
          ((role == 0) ? “Parent” : “Child”));

  return 0;
}
```

The output of the application shown in Listing 14.2 is shown in the following snippet. You see that the child is started and in this case immediately emits some output (its process ID and the first `count` line). The parent and the child then switch off from the GNU/Linux scheduler, each sleeping for one second and emitting a new `count`.

```
# ./smplfork
Child: This is the child process (pid 11024)
Child: At count 0
Parent: This is the parent process (pid 11023)
Parent: At count 0
Parent: At count 1
Child: At count 1
Parent: At count 2
Child: At count 2
Parent: At count 3
Child: At count 3
Parent: At count 4
Child: At count 4
Parent: At count 5
Child: At count 5
Child: Exiting...
Parent: Exiting...
#
```

At the end, you see the `role` variable used to emit the role of the process (parent or child). In this case, whereas the `role` variable was shared between the two processes, after the write occurs, the memory is split, and each process has its own variable, independent of the other. How this occurs is really unimportant. What's important to note is that each process has a copy of its own set of variables.

SYNCHRONIZING WITH THE CREATOR PROCESS

One element of Listing 14.2 was ignored, but this section now digs into it. At line 23, the `wait` function was called within the context of the parent. The `wait` function suspends the parent until the child exits. If the `wait` function is not called by the parent and the child exits, the child becomes what is known as a "zombie" process (neither alive nor dead). It can be problematic to have these processes lying around because of the resources that they waste, so handling child `exit` is necessary. Note that if the parent exits first, the children that have been spawned are inherited by the `init` process.

NOTE

Another way to avoid zombie processes is to tell the parent to ignore child exit signals when they occur. This can be achieved using the `signal` *API function, which is explored in the next section, "Catching a Signal." In any case, after the child has stopped, any system resources that were used by the process are immediately released.*

The first two methods that this chapter discusses for synchronizing the exit of a child process are the `wait` and `waitpid` API functions. The `waitpid` API function provides greater control over the `wait` process; however, for now, this section looks exclusively at the `wait` API function.

The `wait` function suspends the caller (in this case, the parent) awaiting the `exit` of the child. After the child exits, the integer value reference (passed to `wait`) is filled in with the particular exit status. Sample use of the `wait` function, including parsing of the successful status code, is shown in the following code snippet:

```
int status;
pid_t pid;
...
pid = wait( &status );
if ( WIFEXITED(status) ) {
  printf( "Process %d exited normally\n", pid );
}
```

The `wait` function can set other potential status values, which are investigated in the "`wait`" section later in this chapter.

Catching a Signal

A signal is fundamentally an asynchronous callback for processes in GNU/Linux. You can register to receive a signal when an event occurs for a process or register to ignore signals when a default action exists. GNU/Linux supports a variety of signals, which are covered later in this chapter. Signals are an important topic here in process management because they allow processes to communicate with one another.

To catch a signal, you provide a signal handler for the process (a kind of callback function) and the signal that we're interested in for this particular callback. You can now look at an example of registering for a signal. In this example, you register for the `SIGINT` signal. This particular signal identifies that a Ctrl+C was received.

The main program in Listing 14.3 (lines 14–24) begins with registering your callback function (also known as the signal handler). You use the `signal` API function to register your handler (at line 17). You specify first the signal of interest and then the handler function that reacts to the signal. At line 21, you pause, which suspends the process until a signal is received.

The signal handler is shown at Listing 14.3 at lines 6–12. You simply emit a message to `stdout` and then flush it to ensure that it has been emitted. You return from your signal handler, which allows your main function to continue from the `pause` call and exit.

LISTING 14.3 Registering for Catching a Signal (on the CD-ROM at `./source/ch14/sigcatch.c`)

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

void catch_ctlc( int sig_num )
{
  printf( “Caught Control-C\n” );
  fflush( stdout );

  return;
}

int main()
{

  signal( SIGINT, catch_ctlc );

  printf(“Go ahead, make my day.\n”);

  pause();

  return 0;
}
```

Raising a Signal

The previous example illustrated a process receiving a signal. You can also have a process send a signal to another process using the `kill` API function. The `kill` API function takes a process ID (to whom the signal is to be sent) and the signal to send.

Take a look at a simple example of two processes communicating via a signal. This example uses the classic parent/child process creation via `fork` (see Listing 14.4).

At lines 8–13, you declare your signal handler. This handler is very simple, as shown, and simply emits some text to `stdout` indicating that the signal was received, in addition to the process context (identified by the process ID).

The `main` (lines 15–61) is a simple parent/child `fork` example. The parent context (starting at line 25) installs the signal handler and then pauses (awaiting the receipt of a signal). It then continues by awaiting the exit of the child process.

The child context (starting at line 39) sleeps for one second (allowing the parent context to execute and install its signal handler) and then raises a signal. Note that you use the `kill` API function (line 47) to direct the signal to the parent process ID (via `getppid`). The signal you use is `SIGUSR1`, which is a user-definable signal. After the signal has been raised, the child sleeps another two seconds and then exits.

LISTING 14.4 Raising a Signal from a Child to a Parent Process (on the CD-ROM at `./source/ch14/raise.c`)

```
1:        #include <stdio.h>
2:        #include <sys/types.h>
3:        #include <sys/wait.h>
4:        #include <unistd.h>
5:        #include <signal.h>
6:        #include <errno.h>
7:
8:        void usr1_handler( int sig_num )
9:        {
10:
11:         printf( “Parent (%d) got the SIGUSR1\n”, getpid() );
12:
13:       }
14:
15:       int main()
16:       {
17:         pid_t ret;
18:         int   status;
19:         int   role = -1;
20:
21:         ret = fork();
22:
23:         if (ret > 0) {                /* Parent Context */
24:
25:           printf( “Parent: This is the parent process (pid %d)\n”,
26:                    getpid() );
27:
28:           signal( SIGUSR1, usr1_handler );
29:
30:           role = 0;
31:
32:           pause();
33:
34:           printf( “Parent: Awaiting child exit\n” );
```

```
        ret = wait( &status );

    } else if (ret == 0) {         /* Child Context */

        printf( "Child: This is the child process (pid %d)\n",
                 getpid() );

        role = 1;

        sleep( 1 );

        printf( "Child: Sending SIGUSR1 to pid %d\n",
        getppid() );
        kill( getppid(), SIGUSR1 );

        sleep( 2 );

    } else {                        /* Parent Context — Error */

        printf( "Parent: Error trying to fork() (%d)\n",
        errno );

    }

    printf( "%s: Exiting...\n",
             ((role == 0) ? "Parent" : "Child") );

    return 0;
}
```

While this example is probably self-explanatory, looking at its output can be beneficial to understanding exactly what's going on. The output for the application shown in Listing 14.4 is as follows:

```
$ ./raise
Child: This is the child process (pid 14960)
Parent: This is the parent process (pid 14959)
Child: Sending SIGUSR1 to pid 14959
Parent (14959) got the SIGUSR1
Parent: Awaiting child exit
Child: Exiting...
Parent: Exiting...
$
```

You can see that the child performs its first `printf` first (the `fork` gave control of the CPU to the child first). The child then sleeps, allowing the parent to perform its first `printf`, install the signal handler, and then pause awaiting a signal. Now that the parent has suspended, the child can then execute again (after the one-second sleep has finished). It emits its message, indicating that the signal is being raised, and then raises the signal using the `kill` API function. The parent then performs the `printf` within the signal handler (in the context of the parent process as shown by the process ID) and then suspends again awaiting child exit via the `wait` API function. The child process can then execute again, and after the two-second sleep has finished, it exits, releasing the parent from the `wait` call so that it, too, can exit.

It's fairly simple to understand, but it's a powerful mechanism for coordination and synchronization between processes. The entire thread is shown graphically in Figure 14.1. This illustrates the coordination points that exist within your application (shown as dashed horizontal lines from the child to the parent).

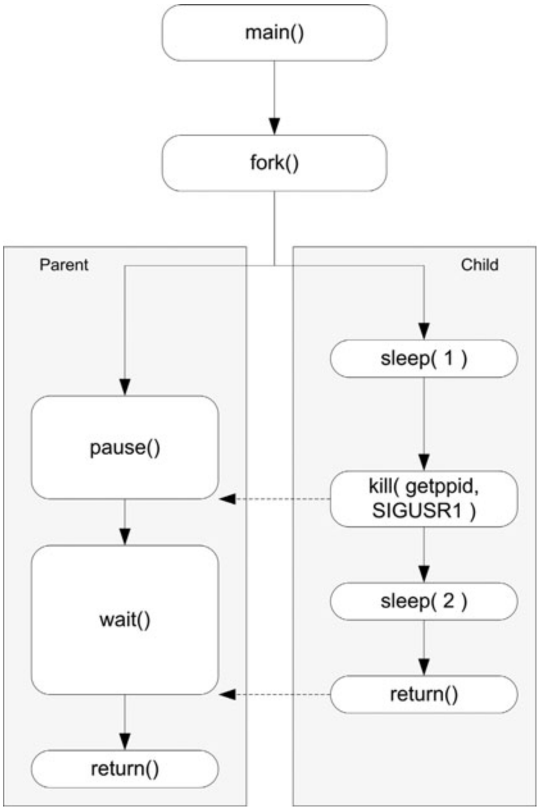

FIGURE 14.1 Graphical illustration of Listing 14.4.

NOTE

If you're raising a signal to yourself (the same process), you can also use the `raise` *API function. This takes the signal to be raised but no process ID argument (because it's automatically* `getpid`*).*

TRADITIONAL PROCESS API

Now that you've looked at a number of different API functions that relate to the GNU/Linux process model, you can now dig further into these functions (and others) and explore them in greater detail. Table 14.1 provides a list of the functions that are explored in the remainder of this section, including their uses.

TABLE 14.1 Traditional Process and Related APIs

API Function	Use
`fork`	Create a new child process.
`wait`	Suspend execution until a child process exits.
`waitpid`	Suspend execution until a specific child process exits.
`signal`	Install a new signal handler.
`pause`	Suspend execution until a signal is caught.
`kill`	Raise a signal to a specified process.
`raise`	Raise a signal to the current process.
`exec`	Replace the current process image with a new process image.
`exit`	Cause normal program termination of the current process.

The remainder of this chapter addresses each of these functions in detail, illustrated in sample applications.

fork

The `fork` API function provides the means to create a new child subprocess from an existing parent process. The new child process is identical to the parent process in almost every way. Some differences include the process ID (a new ID for the child) and that the parent process ID is set to the parent. File locks and signals that are pending to the parent are not inherited by the child process. The prototype for the `fork` function is defined as follows:

```
pid_t fork( void );
```

The fork API function takes no arguments and returns a pid (process identifier). The fork call has a unique structure in that the return value identifies the context in which the process is running. If the return value is zero, then the current process is the newly created child process. If the return value is greater than zero, then the current process is the parent, and the return value represents the process ID of the child. This is illustrated in the following snippet:

```
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
...
pid_t ret;
ret = fork();
if       ( ret > 0 ) {
  /* Parent Process */
  printf( "My pid is %d and my child's is %d\n",
              getpid(), ret );
} else if ( ret == 0 ) {
  /* Child Process */
  printf( "My pid is %d and my parent's is %d\n",
              getpid(), getppid() );
} else {
  /* Parent Process – error */
  printf( "An error occurred in the fork (%d)\n", errno );
}
```

Within the fork() call, the process is duplicated, and then control is returned to the unique process (parent and child). If the return value of fork is less than zero, then an error has occurred. The errno value represents either EAGAIN or ENOMEM. Both errors arise from a lack of available memory.

The fork API function is very efficient in GNU/Linux because of its unique implementation. Rather than copy the page tables for the memory when the fork takes place, the parent and child share the same page tables but are not permitted to write to them. When a write takes place to one of the shared page tables, the page table is copied for the writing process so that it has its own copy. This is called copy-on-write in GNU/Linux and permits the fork to take place very quickly. Only as writes occur to the shared data memory does the segregation of the page tables take place.

wait

The purpose of the wait API function is to suspend the calling process until a child process (created by this process) exits or until a signal is delivered. If the parent isn't currently waiting on the child to exit, the child exits, and the child process becomes a zombie process.

The `wait` function provides an asynchronous mechanism as well. If the child process exits before the parent has had a chance to call `wait`, then the child becomes a zombie. However, it is then freed after `wait` is called. The `wait` function, in this case, returns immediately.

The prototype for the `wait` function is defined as follows:

```
pid_t wait( int *status );
```

The `wait` function returns the `pid` value of the child that exited, or –1 if an error occurred. The `status` variable (whose reference is passed into `wait` as its only argument) returns status information about the child exit. This variable can be evaluated using a number of macros. These macros are listed in Table 14.2.

TABLE 14.2 Macro Functions to Evaluate `wait` Status

Macro	Description
`WIFEXITED`	Nonzero if the child exited normally
`WEXITSTATUS`	Returns the `exit` status of the child
`WIFSIGNALED`	Returns `true` if child exited because of a signal that wasn't caught by the child
`WTERMSIG`	Returns the signal number that caused the child to exit (relevant only if `WIFSIGNALED` is `true`)

The general form of the status evaluation macro is demonstrated in the following code snippet:

```
pid = wait( &status );
if        ( WIFEXITED(status) ) {
  printf( "Child exited normally with status %d\n",
                WEXITSTATUS(status) );
} else if ( WIFSIGNALED(status) ) {
  printf( "Child exited by signal with status %d\n",
                WTERMSIG(status) );
}
```

In some cases, you're not interested in the `exit` status of your child processes. In the `signal` API function discussion, you can see a way to ignore this status so that `wait` does not need to be called by the parent to avoid child zombie processes.

waitpid

Whereas the wait API function suspends the parent until a child exits (any child), the waitpid API function suspends until a specific child exits. The waitpid function provides some other capabilities, which are explored here. The waitpid function prototype is defined as follows:

```
pid_t waitpid( pid_t pid, int *status, int options );
```

The return value for waitpid is the process identifier for the child that exited. The return value can also be zero if the options argument is set to WNOHANG and no child process has exited (returns immediately).

The arguments to waitpid are a pid value, a reference to a return status, and a set of options. The pid value can be a child process ID or other values that provide different behaviors. Table 14.3 lists the possible pid values for waitpid.

TABLE 14.3 pid Arguments for waitpid

Value	Description
> 0	Suspend until the child identified by the pid value has exited
0	Suspend until any child exits whose group ID matches that of the calling process
−1	Suspend until any child exits (identical to the wait function)
< −1	Suspend until any child exits whose group ID is equal to the absolute value of the pid argument

The status argument for waitpid is identical to the wait function, except that two new status macros are possible (see Table 14.4). These macros are seen only if the WUNTRACED option is specified.

TABLE 14.4 Extended Macro Functions for waitpid

Macro	Description
WIFSTOPPED	Returns true if the child process is currently stopped
WSTOPSIG	Returns the signal that caused the child to stop (relevant only if WIFSTOPPED was nonzero)

The final argument to waitpid is the options argument. Two options are available: WNOHANG and WUNTRACED. WNOHANG, as discussed, avoids suspension of the parent process and returns only if a child has exited. The `WUNTRACED` option returns for children that have been stopped and not yet reported.

Now it's time to take a look at some examples of the `waitpid` function. In the first code snippet, you `fork` off a new child process and then await it explicitly (rather than as with the `wait` method that waits for any child).

```
pid_t child_pid, ret;
int status;
...
child_pid = fork();
if (child_pid == 0) {
  // Child process...
} else if (child_pid > 0) {
  ret = waitpid( child_pid, &status, 0 );
  /* Note ret should equal child_pid on success */
  if ( WIFEXITED(status) ) {
    printf( "Child exited normally with status %d\n",
                 WEXITSTATUS(status) );
  }
}
```

In this example, you `fork` off your child and then use `waitpid` with the child's process ID. Note here that you can use the status macro functions that were defined with `wait` (as demonstrated with `WIFEXITED`). If you don't want to wait for the child, you can specify `WNOHANG` as an option. This requires you to call `waitpid` periodically to handle the child `exit`:

```
ret = waitpid( child_pid, &status, WNOHANG );
```

The following line awaits a child process exiting the defined group. Note that you negate the group ID in the call to `waitpid`. Also notable is passing `NULL` as the status reference. In this case, you're not interested in getting the child's `exit` status. In any case, the `return` value is the process ID for the child process that exited.

```
pid_t group_id;
...
ret = waitpid( -group_id, NULL, 0 );
```

signal

The `signal` API function allows you to install a signal handler for a process. The signal handler passed to the `signal` API function has the following form:

```
void signal_handler( int signal_number );
```

After it is installed, the function is called for the process when the particular signal is raised to the process. The prototype for the `signal` API function is defined as follows:

```
sighandler_t signal( int signum, sighandler_t handler );
```

where the `sighandler_t typedef` is as follows:

```
typedef void (*sighandler_t)(int);
```

The `signal` function returns the previous signal handler that was installed, which allows the new handler to chain the older handlers together (if necessary).

A process can install handlers to catch signals, and it can also define that signals should be ignored (`SIG_IGN`). To ignore a signal for a process, the following code snippet can be used:

```
signal( SIGCHLD, SIG_IGN );
```

After this particular code is executed, it is not necessary for a parent process to wait for the child to exit using `wait` or `waitpid`.

Signal handlers for a process can be of three different types. They can be ignored (via `SIG_IGN`), the default handler for the particular signal type (`SIG_DFL`), or a user-defined handler (installed via `signal`).

A large number of signals exist for GNU/Linux. They are provided in Tables 14.5–14.8 with their meanings. The signals are split into four groups, based upon default action for the signal.

TABLE 14.5 GNU/Linux Signals That Default to Terminate

Signal	Description
SIGHUP	Hang up—commonly used to restart a task
SIGINT	Interrupt from the keyboard
SIGKILL	Kill signal
SIGUSR1	User-defined signal
SIGUSR2	User-defined signal
SIGPIPE	Broken pipe (no reader for write)
SIGALRM	Timer signal (from API function `alarm`)
SIGTERM	Termination signal
SIGPROF	Profiling timer expired

TABLE 14.6 GNU/Linux Signals That Default to Ignore

Signal	Description
SIGCHLD	Child stopped or terminated
SIGCLD	Same as SIGCHLD
SIGURG	Urgent data on a socket

TABLE 14.7 GNU/Linux Signals That Default to Stop

Signal	Description
SIGSTOP	Stop process
SIGTSTP	Stop initiated from TTY
SIGTTIN	Background process has TTY input
SIGTTOU	Background process has TTY output

TABLE 14.8 GNU/Linux Signals That Default to Core Dump

Signal	Description
SIGQUIT	`quit` signal from keyboard
SIGILL	Illegal instruction encountered
SIGTRAP	Trace or breakpoint trap
SIGABRT	Abort signal (from API function `abort`)
SIGIOT	IOT trap, same as SIGABRT
SIGBUS	Bus error (invalid memory access)
SIGFPE	Floating-point exception
SIGSEGV	Segment violation (invalid memory access)

The first group (terminate) lists the signals whose default action is to terminate the process. The second group (ignore) lists the signals for which the default action is to ignore the signal. The third group (stop) stops the process (suspends rather than terminates). Finally, the fourth group (core) lists those signals whose action is to both terminate the process and perform a core dump (generate a core dump file).

It's important to note that the `SIGSTOP` and `SIGKILL` signals cannot be ignored or caught by the application. One other signal not categorized in the preceding information is the `SIGCONT` signal, which is used to continue a process if it was previously stopped.

GNU/Linux also supports 32 real-time signals (of POSIX 1003.1-2001). The signals are numbered from 32 (`SIGRTMIN`) up to 63 (`SIGRTMAX`) and can be sent using the `sigqueue` API function. The receiving process must use `sigaction` to install the signal handler (discussed later in this chapter) to collect other data provided in this signaling mechanism.

Now it's time to look at a simple application that installs a signal handler at the parent, which is inherited by the child (see Listing 14.5). In this listing, you first declare a signal handler (lines 8–13) that is installed by the parent prior to the `fork` (at line 21). Installing the handler prior to the `fork` means that the child inherits this signal handler as well.

After the `fork` (at line 23), the parent and child context emit an identification string to `stdout` and then call the `pause` API function (which suspends each process until a signal is received). When a signal is received, the signal handler prints out the context in which it caught the signal (via `getpid`) and then either exits (child process) or awaits the exit of the child (parent process).

LISTING 14.5 Signal Demonstration with a Parent and Child Process (on the CD-ROM at `./source/ch14/sigtest.c`)

```
1:    #include <stdio.h>
2:    #include <sys/types.h>
3:    #include <sys/wait.h>
4:    #include <unistd.h>
5:    #include <signal.h>
6:    #include <errno.h>
7:
8:    void usr1_handler( int sig_num )
9:    {
10:
11:     printf( “Process (%d) got the SIGUSR1\n”, getpid() );
12:
13:   }
14:
15:   int main()
16:   {
17:     pid_t ret;
18:     int   status;
```

```
    int   role = -1;

    signal( SIGUSR1, usr1_handler );

    ret = fork();

    if (ret > 0) {                /* Parent Context */

      printf( "Parent: This is the parent process (pid %d)\n",
               getpid() );

      role = 0;

      pause();

      printf( "Parent: Awaiting child exit\n" );
      ret = wait( &status );

    } else if (ret == 0) {        /* Child Context */

      printf( "Child: This is the child process (pid %d)\n",
               getpid() );

      role = 1;

      pause();

    } else {                     /* Parent Context — Error */

      printf( "Parent: Error trying to fork() (%d)\n",
      errno );

    }

    printf( "%s: Exiting...\n",
             ((role == 0) ? "Parent" : "Child") );

    return 0;
  }
```

Now consider the sample output for this application to better understand what happens. Note that neither the parent nor the child raises any signals to each other. This example takes care of sending the signal at the command line, using the `kill` command.

```
# ./sigtest &
[1] 20152
# Child: This is the child process (pid 20153)
Parent: This is the parent process (pid 20152)

# kill -10 20152
Process (20152) got the SIGUSR1
Parent: Awaiting child exit
# kill -10 20153
Process (20153) got the SIGUSR1
Child: Exiting...
Parent: Exiting...
#
```

You begin by running the application (called `sigtest`) and placing it in the background (via the `&` symbol). You see the expected outputs from the child and parent processes identifying that the `fork` has occurred and that both processes are now active and awaiting signals at the respective `pause` calls. You use the `kill` command with the signal of interest (`–10`, or `SIGUSR1`) and the process identifier to which to send the signal. In this case, you send the first `SIGUSR1` to the parent process (20152). The parent immediately identifies receipt of the signal via the signal handler, but note that it executes within the context of the parent process (as identified by the process ID of 20152). The parent then returns from the `pause` function and awaits the exit of the child via the `wait` function. You then send another `SIGUSR1` signal to the child using the `kill` command. In this case, you direct the `kill` command to the child by its process ID (20153). The child also indicates receipt of the signal by the signal handler and in its own context. The child then exits and permits the parent to return from the `wait` function and exit also.

Despite the simplicity of the signals mechanism, it can be a powerful method to communicate with processes in an asynchronous fashion.

pause

The `pause` function suspends the calling process until a signal is received. After the signal is received, the calling process returns from the `pause` function, permitting it to continue. The prototype for the `pause` API function is as follows:

```
int pause( void );
```

If the process has installed a signal handler for the signal that was caught, then the `pause` function returns after the signal handler has been called and returns.

kill

The `kill` API function raises a signal to a process or set of processes. A return of zero indicates that the signal was successfully sent, otherwise –1 is returned. The `kill` function prototype is as follows:

```
int kill( pid_t pid, int sig_num );
```

The `sig_num` argument represents the signal to send. The `pid` argument can be a variety of different values (as shown in Table 14.9).

TABLE 14.9 Values of `pid` Argument for `kill` Function

pid	**Description**
0	Signal sent to the process defined by `pid`
0	Signal sent to all processes within the process group
1	Signal sent to all processes (except for the `init` process)
–1	Signal sent to all processes within the process group defined by the absolute value of `pid`

Some simple examples of the `kill` function follow. You can send a signal to yourself using the following code snippet:

```
kill( getpid(), SIGHUP );
```

The process group enables you to collect a set of processes together that can be signaled together as a group. API functions such as `getpgrp` (get process group) and `setpgrp` (set process group) can be used to read and set the process group identifier. You can send a signal to all processes within a defined process group as follows:

```
kill( 0, SIGUSR1 );
```

or to another process group as follows:

```
pid_t group;
...
kill( -group, SIGUSR1 );
```

You can also mimic the behavior of sending to the current process group by identifying the group and then passing the negative of this value to signal:

```
pid_t group = getpgrp();
...
kill( -group, SIGUSR1 );
```

Finally, you can send a signal to all processes (except for `init`) using the –1 `pid` identifier. This, of course, requires that you have permission to do this.

```
kill( -1, SIGUSR1 );
```

raise

The `raise` API function can be used to send a specific signal to the current process (the process context in which the `raise` function is called). The prototype for the `raise` function is as follows:

```
int raise( int sig_num );
```

The `raise` function is a constrained version of the `kill` API function that targets only the current process (`getpid()`).

exec Variants

The `fork` API function provided a mechanism to split an application into separate parent and child processes, sharing the same code but potentially serving different roles. The `exec` family of functions replaces the current process image altogether.

NOTE

The `exec` *function starts a new program, replacing the current process, while retaining the current* `pid`.

The prototypes for the variants of `exec` are provided here:

```
int execl( const char *path, const char *arg, ... );
int execlp( const char *path, const char *arg, ... );
int execle( const char *path, const char *arg, ...,
              char * const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv[],
           char *const envp[] );
```

One of the notable differences between these functions is that one set takes a list of parameters (`arg0`, `arg1`, and so on) and the other takes an `argv` array. The `path` argument specifies the program to run, and the remaining parameters specify the arguments to pass to the program.

The `exec` commands permit the current process context to be replaced with the program (or command) specified as the first argument. Take a look at a quick example of `execcl` to achieve this:

```
execl( "/bin/ls", "ls", "-la", NULL );
```

This command replaces the current process with the `ls` image (list directory). You specify the command to execute as the first argument (including its path). The second argument is the command again (recall that `arg0` of the `main` program call is the name of the program). The third argument is an option that you pass to `ls`, and finally, you identify the end of your list with a `NULL`. Invoking an application that performs this command results in an `ls -la`.

The important item to note here is that the current process context is replaced by the command requested via `execl`. Therefore, when the preceding command is successfully executed, it never returns.

One additional item to note is that `execl` includes the absolute path to the command. If you choose to execute `execlp` instead, the full path is not required because the parent's `PATH` definition is used to find the command.

One interesting example of `execlp` is its use in creating a simple shell (on top of an existing shell). You support only simple commands within this shell (those that take no arguments). See Listing 14.6 for an example.

LISTING 14.6 Simple Shell Interpreter Using `execlp` (on the CD-ROM at `./source/ch14/simpshell.c`)

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

#define MAX_LINE        80

int main()
{
  int status;
  pid_t childpid;
```

```
14:      char cmd[MAX_LINE+1];
15:      char *sret;
16:
17:      while (1) {
18:
19:        printf("mysh>");
20:
21:        sret = fgets( cmd, sizeof(cmd), stdin );
22:
23:        if (sret == NULL) exit(-1);
24:
25:        cmd[ strlen(cmd)-1] = 0;
26:
27:        if (!strncmp(cmd, "bye", 3)) exit(0);
28:
29:        childpid = fork();
30:
31:        if (childpid == 0) {
32:
33:          execlp( cmd, cmd, NULL );
34:
35:        } else if (childpid > 0) {
36:
37:          waitpid( childpid, &status, 0 );
38:
39:        }
40:
41:        printf("\n");
42:
43:      }
44:
45:      return 0;
46:    }
```

This shell interpreter is built around the simple parent/child `fork` application. The parent forks off the child (at line 29) and then awaits completion. The child takes the command read from the user (at line 21) and executes this using `execlp` (line 33). You simply specify the command as the command to execute and also include it for `arg0` (second argument). The `NULL` terminates the argument list; in this case no arguments are passed for the command. The child process never returns, but its `exit` status is recognized by the parent at the `waitpid` function (line 37).

As the user types in commands, they are executed via `execlp`. Typing in the command `bye` causes the application to exit.

Because no arguments are passed to the command (via `execlp`), the user can type in only commands and no arguments. Any arguments that are provided are simply ignored by the interpreter.

A sample execution of this application is shown here:

```
$ ./simpshell
mysh>date
Sat Apr 24 13:47:48 MDT 2004
mysh>ls
simpshell    simpshell.c
mysh>bye
$
```

You can see that after executing the shell, the prompt is displayed, indicating that commands can be entered. The `date` command is entered first, which provides the current date and time. Next, you do an `ls`, which gives the contents of the current directory. Finally, you exit the shell using the `bye` internal command.

Now take a look at one final `exec` variant as a way to explore the argument and environment aspects of a process. The `execve` variant allows an application to provide a command with a list of command-line arguments (as a vector) as well as an environment for the new process (as a vector of environment variables). Now take a look back at the `execve` prototype:

```
int execve( const char *filename, char *const argv[],
            char *const envp[] );
```

The `filename` argument is the program to execute, which must be a binary executable or a script that includes the `#!` interpreter spec at the top of the file. The `argv` argument is an array of arguments for the command, with the first argument being the command itself (the same as with the `filename` argument). Finally, the `envp` argument is an array of key/value strings containing environment variables. Consider the following simple example that retrieves the environment variables through the `main` function (on the CD-ROM at `./source/ch14/sigenv.c`):

```
#include <stdio.h>
#include <unistd.h>
int main( int argc, char *argv[], char *envp[] )
{
  int ret;
  char *args[]={ "ls", "-la", NULL };
```

```
    ret = execve( "/bin/ls", args, envp );
    fprintf( stderr, "execve failed\n" );
    return 0;
  }
```

The first item to note in this example is the `main` function definition. You use a variant that passes in a third parameter that lists the environment for the process. This can also be gathered by the program using the special `environ` variable, which has the following definition:

```
extern char *environ[];
```

NOTE

POSIX systems do not support the `envp` *argument to* `main`*, so it's best to use the* `environ` *variable.*

You specify your argument vector (`args`), which contains your command name and arguments, terminated by a `NULL`. This is provided as the argument vector to `execve`, along with the environment (passed in through the `main` function). This particular example simply performs an `ls` operation (by replacing the process with the `ls` command). Note also that you provide the `-la` option.

You can also specify your own environment similar to the `args` vector. For example, the following specifies a new environment for the process:

```
char *envp[] = { "PATH=/bin", "FOO=99", NULL };
...
ret = execve( command, args, envp );
```

The `envp` variable provides the set of variables that define the environment for the newly created process.

alarm

The `alarm` API function can be very useful to time out other functions. The `alarm` function works by raising a `SIGALRM` signal after the number of seconds passed to `alarm` has expired. The function prototype for `alarm` is as follows:

```
unsigned int alarm( unsigned int secs );
```

The user passes in the number of seconds to wait before sending the `SIGALRM` signal. The `alarm` function returns zero if no alarm was previously scheduled; otherwise, it returns the number of seconds pending on the previous alarm.

Here's an example of `alarm` to kill the current process if the user isn't able to enter a password in a reasonable amount of time (see Listing 14.7). At line 18, you install your signal handler for the `SIGALRM` signal. The signal handler is for the `wakeup` function (lines 6–9), which simply raises the `SIGKILL` signal. This terminates the application. You then emit the message to enter the password within three seconds and try to read the password from the keyboard (`stdin`). If the `read` call succeeds, you disable the alarm (by calling `alarm` with an argument of zero). The `else` portion of the test (line 30) checks the user password and continue. If the alarm times out, a `SIGALRM` is generated, resulting in a `SIGKILL` signal, which terminates the program.

LISTING 14.7 Sample Use of `alarm` and Signal Capture (on the CD-ROM at `./source/ch14/alarm.c`)

```
1:       #include <stdio.h>
2:       #include <unistd.h>
3:       #include <signal.h>
4:       #include <string.h>
5:
6:       void wakeup( int sig_num )
7:       {
8:         raise(SIGKILL);
9:       }
10:
11:      #define MAX_BUFFER     80
12:
13:      int main()
14:      {
15:        char buffer[MAX_BUFFER+1];
16:        int ret;
17:
18:        signal( SIGALRM, wakeup );
19:
20:        printf(“You have 3 seconds to enter the password\n”);
21:
22:        alarm(3);
23:
24:        ret = read( 0, buffer, MAX_BUFFER );
25:
26:        alarm(0);
27:
28:        if (ret == -1) {
29:
```

```
        } else {

            buffer[strlen(buffer)-1] = 0;
            printf("User entered %s\n", buffer);

        }

    }
```

exit

The exit API function terminates the calling process. The argument passed to exit is returned to the parent process as the status of the parent's wait or waitpid call. The function prototype for exit is as follows:

```
void exit( int status );
```

The process calling exit also raises a SIGCHLD to the parent process and frees the resources allocated by the process (such as open file descriptors). If the process has registered a function with atexit or on_exit, these are called (in the reverse order to their registration).

This call is very important because it indicates success or failure to the shell environment. Scripts that rely on a program's exit status can behave improperly if the application does not provide an adequate status. This call provides that linkage to the scripting environment. Returning zero to the script indicates a TRUE or SUCCESS indication.

POSIX SIGNALS

Before this discussion of process-related functions ends, you need to take a quick look at the POSIX signal APIs. The POSIX-compliant signals were introduced first in BSD and provide a portable API over the use of the signal API function. Have a look at a multiprocess application that uses the sigaction function to install a signal handler. The sigaction API function has the following prototype:

```
#include <signal.h>
int sigaction( int signum,
               const struct sigaction *act,
               struct sigaction *oldact );
```

signum is the signal for which you're installing the handler, act specifies the action to take for signum, and oldact is used to store the previous action. The sigaction structure contains a number of elements that can be configured:

```
struct sigaction {
    void (*sa_handler)( int );
    void (*sa_sigaction)( int, siginfo_t *, void * );
    sigset_t sa_mask;
    int sa_flags;
};
```

The `sa_handler` is a traditional signal handler that accepts a single argument (and `int` represents the signal). The `sa_sigaction` is a more refined version of a signal handler. The first `int` argument is the signal, and the third `void*` argument is a context variable (provided by the user). The second argument (`siginfo_t`) is a special structure that provides more detailed information about the signal that was generated:

```
siginfo_t {
  int         si_signo;     /* Signal number */
  int         si_errno;     /* Errno value */
  int         si_code;      /* Signal code */
  pid_t       si_pid;       /* Pid of signal sending process */
  uid_t       si_uid;       /* User id of signal sending process */
  int         si_status;    /* Exit value or signal */
  clock_t     si_utime;     /* User time consumed */
  clock_t     si_stime;     /* System time consumed */
  sigval_t    si_value      /* Signal value */
  int         si_int;       /* POSIX.1b signal */
  void *      si_ptr        /* POSIX.1b signal */
  void *      si_addr       /* Memory location which caused fault */
  int         si_band;      /* Band Event */
  int         si_fd;        /* File Descriptor */
}
```

One of the interesting items to note from `siginfo_t` is that with this API, you can identify the source of the signal (`si_pid`). The `si_code` field can be used to identify how the signal was raised. For example, if its value is `SI_USER`, then it was raised by a `kill`, `raise`, or `sigsend` API function. If its value is `SI_KERNEL`, then it was raised by the kernel. `SI_TIMER` indicates that a timer expired and resulted in the signal generation.

The `si_signo`, `si_errno`, and `si_code` are set for all signals. The `si_addr` field (indicating the memory location where the fault occurred) is set for `SIGILL`, `SIGFPE`, `SIGSEGV`, and `SIGBUS`. The `sigaction` main page identifies which fields are relevant for which signals.

The sa_flags argument of sigaction allows a modification of the behavior of the sigaction function. For example, if you provide SA_SIGINFO, then the sigaction uses the sa_sigaction field to identify the signal handler instead of sa_handler. Flag SA_ONESHOT can be used to restore the signal handler to the prior state after the signal handler has been called once. The SA_NOMASK (or SA_NODEFER) flag can be used to not inhibit the reception of the signal while in the signal handler (use with care).

A sample function is provided in Listing 14.8. The only real difference you see here from other examples is that sigaction is used at line 49 to install your signal handler. You create a sigaction structure at line 42, then initialize it with your function at line 48, and also identify that you're using the new sigaction handler via the SA_SIGINFO flag at line 47. When your signal finally fires (at line 34 in the parent process), your signal handler emits the originating process ID at line 12 (using the si_pid field of the siginfo reference).

LISTING 14.8 Simple Application Illustrating sigaction for Signal Installation (on the CD-ROM at ./source/ch14/posixsig.c)

```
1:       #include <sys/types.h>
2:       #include <sys/wait.h>
3:       #include <signal.h>
4:       #include <stdio.h>
5:       #include <unistd.h>
6:       #include <errno.h>
7:
8:       static int stopChild = 0;
9:
10:      void sigHandler( int sig, siginfo_t *siginfo, void *ignore )
11:      {
12:        printf("Got SIGUSR1 from %d\n", siginfo->si_pid);
13:        stopChild=1;
14:
15:        return;
16:      }
17:
18:      int main()
19:      {
20:        pid_t ret;
21:        int   status;
22:        int   role = -1;
23:
24:        ret = fork();
25:
```

```
      if (ret > 0) {

        printf("Parent: This is the parent process (pid %d)\n",
                getpid());

        /* Let the child init */
        sleep(1);

        kill( ret, SIGUSR1 );

        ret = wait( &status );

        role = 0;

      } else if (ret == 0) {

        struct sigaction act;

        printf("Child: This is the child process (pid %d)\n",
                getpid());

        act.sa_flags = SA_SIGINFO;
        act.sa_sigaction = sigHandler;
        sigaction( SIGUSR1, &act, 0 );

        printf("Child Waiting...\n");
        while (!stopChild);

        role = 1;

      } else {

        printf("Parent: Error trying to fork() (%d)\n", errno);

      }

      printf("%s: Exiting...\n",
              ((role == 0) ? "Parent" : "Child"));

      return 0;
    }
```

The `sigaction` function provides a more advanced mechanism for signal handling, in addition to greater portability. For this reason, `sigaction` should be used over `signal`.

SYSTEM COMMANDS

This section takes a look at a few of the GNU/Linux commands that work with the previously mentioned API functions. It looks at commands that permit you to inspect the process list and send a signal to a process or to an entire process group.

ps

The `ps` command provides a snapshot in time of the current set of processes active on a given system. The `ps` command takes a large variety of options; this section explores a few.

In the simplest form, you can type `ps` at the keyboard to see a subset of the processes that are active:

```
$ ps
  PID TTY          TIME CMD
22001 pts/0    00:00:00 bash
22186 pts/0    00:00:00 ps
$
```

First, you see your `bash` session (your own process) and your `ps` command process (every command in GNU/Linux is executed within its own subprocess). You can see all of the processes running using the `-a` option (this list is shortened for brevity):

```
$ ps -a
  PID TTY          TIME CMD
    1 ?        00:00:05 init
    2 ?        00:00:00 keventd
    3 ?        00:00:00 kapmd
    4 ?        00:00:00 ksoftirqd_CPU0
...
22001 pts/0    00:00:00 bash
22074 ?        00:00:00 sendmail
22189 pts/0    00:00:00 ps
$
```

In this example, you see a number of other processes including the mother of all processes (`init`, process ID 1) and assorted kernel threads. If you want to see only those processes that are associated with your user, you can accomplish this with the `–User` option:

```
$ ps –User mtj
  PID TTY          TIME CMD
22000 ?        00:00:00 sshd
22001 pts/0    00:00:00 bash
22190 pts/0    00:00:00 ps
$
```

Another very useful option is `-H`, which tells you the process hierarchy. In the next example, you request all processes for user `mtj` but then also request their hierarchy (parent/child relationships):

```
$ ps –User mtj -H
  PID TTY          TIME CMD
22000 ?        00:00:00 sshd
22001 pts/0    00:00:00   bash
22206 pts/0    00:00:00     ps
#
```

Here you see that the base process is an `sshd` session (because you are connected to this server via the secure shell). This is the parent of the `bash` session, which in turn is the parent of the `ps` command that you just executed.

The `ps` command can be very useful, especially when you're interested in finding your process identifiers to kill a process or send it a signal.

top

The `top` command is related to `ps`, but `top` runs in real time and lists the activity of the processes for the given CPU. In addition to the process list, you can also see statistics about the CPU (number of processes, number of zombies, memory used, and so on). You're obviously in need of a memory upgrade here (only 4 MB free). This sample list has again been shortened for brevity.

```
 19:27:49  up 79 days, 10:04,  2 users,  load average: 0.00, 0.00, 0.00
47 processes: 44 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:   0.0% user   0.1% system   0.0% nice   0.0% iowait  99.8% idle
Mem:   124984k av,  120892k used,    4092k free,       0k shrd,   52572k buff
                     79408k actv,       4k in_d,     860k in_c
```

```
Swap:  257032k av,    5208k used,  251824k free                  37452k
                                                                 cached
  PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
22226 mtj        15   0  1132 1132   868 R     0.1  0.9   0:00   0 top
    1 root       15   0   100   76    52 S     0.0  0.0   0:05   0 init
    2 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 keventd
    3 root       15   0     0    0     0 RW    0.0  0.0   0:00   0 kapmd
    4 root       34  19     0    0     0 SWN   0.0  0.0   0:00   0 ksoftirqd_
                                                                   CPU0
...
 1708 root       15   0   196    4     0 S     0.0  0.0   0:00   0 login
 1709 root       15   0   284    4     0 S     0.0  0.0   0:00   0 bash
22001 mtj        15   0  1512 1512  1148 S     0.0  1.2   0:00   0 bash
```

The rate of sampling can also be adjusted for `top`, in addition to a number of other options (see the `top` man page for more details).

kill

The `kill` command, like the `kill` API function, allows you to send a signal to a process. You can also use it to list the signals that are relevant for the given processor architecture. For example, if you want to see the signals that are available for the given processor, you use the `-l` option:

```
# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-14 50) SIGRTMAX-13
51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10 54) SIGRTMAX-9
55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6  58) SIGRTMAX-5
59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2  62) SIGRTMAX-1
63) SIGRTMAX
#
```

For a running process, you can send a signal as follows. In this example, you send the `SIGSTOP` signal to the process identified by the process ID 23000.

```
# kill -s SIGSTOP 23000
```

This places the process in the `STOPPED` state (not running). You can start the process up again by giving it the `SIGCONT` signal, as follows:

```
# kill -s SIGCONT 23000
```

Like the `kill` API function, you can signal an entire process group by providing a `pid` of 0. Similarly, all processes within the process group can be sent a signal by sending the negative of the process group.

SUMMARY

This chapter explored the traditional process API provided in GNU/Linux. You investigated process creation with `fork`, validating the status return of `fork`, and various process-related API functions such as `getpid` (get process ID) and `getppid` (get parent process ID). The chapter then looked at process support functions such as `wait` and `waitpid` and the signal mechanism that permits processes to communicate with one another. Finally, you looked at a number of GNU/Linux commands that enable you to review active processes and also the commands to signal them.

PROC FILESYSTEM

The `/proc` filesystem is the root source of information about the processes within a GNU/Linux system. Within `/proc`, you'll find a set of directories with numbered filenames. These numbers represent the process IDs (`pids`) of active processes within the system. The root-level view of `/proc` is provided in the following:

```
# ls /proc
1      4     5671  7225  9780      crypto       kcore      stat
10     4307  6     7255  9783      devices      key-users  swaps
19110  4524  6265  7265  9786      diskstats    kmsg       sys
2      5     6387  7360  9787      dma          loadavg    sysrq-
trigger
2132   5009  6416  8134  9788      driver       locks      sysvipc
21747  5015  6446  9     9789      execdomains  mdstat     tty
```

```
21748  5312  6671  94    9790       fb           meminfo     uptime
21749  5313  6672  95    9791       filesystems  misc        version
21751  5340  6673  9650  9795       fs           modules     vmstat
2232   5341  6674  9684  9797       ide          mounts      zoneinfo
24065  5342  6675  9688  acpi       interrupts   mtrr
2623   5343  683   9689  buddyinfo  iomem        net
3      5344  6994  9714  bus        ioports      partitions
32618  5560  7     9715  cmdline    irq          self
3651   5670  7224  9773  cpuinfo    kallsyms     slabinfo
#
```

Each pid directory presents a hierarchy of information about that process including the command line that started it, a symlink to the root filesystem (which can be different from the current root if the executable was chrooted), a symlink to the directory (current working directory) where the process was started, and others. The following is a look at a pid directory hierarchy:

```
# ls /proc/1
attr     cpuset   exe   mem      oom_score  smaps  status
auxv     cwd      fd    mounts   root       stat   task
cmdline  environ  maps  oom_adj  seccomp    statm  wchan
#
```

Recall that pid 1 is the first process to execute and is always the init process. You can view this from the status file that contains basic information about the process including its state, memory usage, signal masks, etc.

```
# cat status
Name:   init
State:  S (sleeping)
SleepAVG:       88%
Tgid:   1
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups:
...
#
```

The /proc filesystem also contains a large number of other nonprocess-specific elements, some of which can be written to change the behavior of the overall operating system. Many utilities use information from the /proc filesystem to present data to the user (for example, the ps command uses /proc to get its process list).

References

GNU/Linux signal and sigaction man pages.

API Summary

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
pid_t fork( void );
pid_t wait( int *status );
pid_t waitpid( pid_t pid, int *status, int options );
sighandler_t signal( int signum, sighandler_t handler );
int pause( void );
int kill( pid_t pid, int sig_num );
int raise( int sig_num );
int execl( const char *path, const char *arg, ... );
int execlp( const char *path, const char *arg, ... );
int execle( const char *path, const char *arg, ...,
              char * const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv[],
          char *const envp[] );
unsigned int alarm( unsigned int secs );
void exit( int status );
int sigaction( int signum,
           const struct sigaction *act,
           struct sigaction *oldact );
```

15 POSIX Threads (`pthreads`) Programming

In This Chapter

- Threads and Processes
- Creating Threads
- Synchronizing Threads
- Communicating Between Threads
- POSIX Signals API
- Threaded Application Development Topics

INTRODUCTION

Multithreaded applications are a useful paradigm for system development because they offer many facilities not available to traditional GNU/Linux processes. This chapter explores `pthreads` programming and the functionality provided by the `pthreads` API.

NOTE

The 2.4 GNU/Linux kernel POSIX thread library was based upon the Linux-Threads implementation (introduced in 1996), which was built on the existing GNU/Linux process model. The 2.6 kernel utilizes the new Native POSIX Thread Library, or NPTL (introduced in 2002), which is a higher performance implementation with numerous advantages over the older component. For example, NPTL provides real thread groups (within a process), compared to one thread per process in the prior model. This chapter outlines those differences when they are useful to know.

To know which `pthreads` *library is being used, issue the following command:*

```
$ getconf GNU_LIBPTHREAD_VERSION
```

This provides either LinuxThreads or NPTL, each with a version number.

What's a Thread?

To define a thread, you need to look back at Linux processes to understand their makeup. Both processes and threads have control flows and can run concurrently, but they differ in some very distinct ways. Threads, for example, share data, where processes explicitly don't. When a process is forked (recall from Chapter 13, "Introduction to Sockets Programming"), a new process is created with its own globals and stack (see Figure 15.1). When a thread is created, the only new element created is a stack that is unique for the thread (see Figure 15.2). The code and global data are common between the threads. This is advantageous, but the shared nature of threads can also be problematic. This chapter investigates this later.

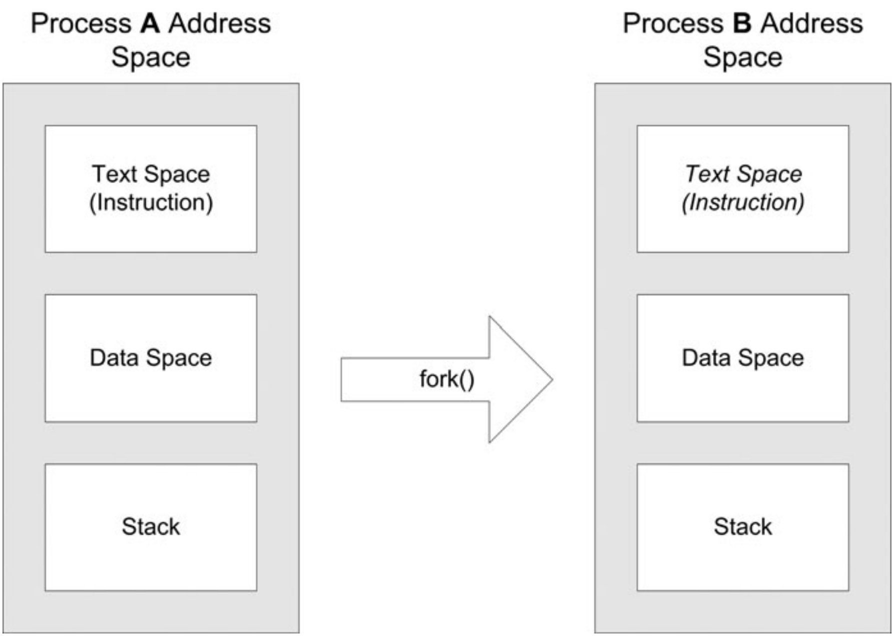

FIGURE 15.1 Forking a new process.

A GNU/Linux process can create and manage numerous threads. Each thread is identified by a thread identifier that is unique for every thread in a system. Each thread also has its own stack (as shown in Figure 15.2) and also a unique context (program counter, save registers, and so forth). But because the data space is shared by threads, they share more than just user data. For example, file descriptors for open files or sockets are shared also. Therefore, when a multithreaded application uses a socket or file, the access to the resource must be protected against multiple accesses. This chapter looks at methods for achieving that.

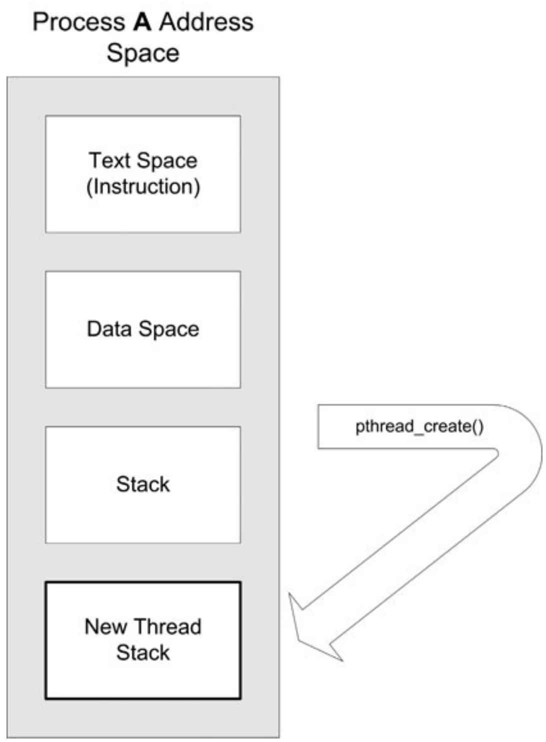

FIGURE 15.2 Creating a new thread.

NOTE

While writing multithreaded applications can be easier in some ways than traditional process-based applications, you do encounter problems you need to understand. The shared data aspect of threads is probably the most difficult to design around, but it is also powerful and can lead to simpler applications with higher performance. The key is to strongly consider shared data while developing threaded applications. Another important consideration is that serious multithreaded application development needs to utilize the 2.6 kernel rather than the 2.4 kernel (given the new NPTL threads implementation).

Thread Function Basics

The APIs discussed thus far follow a fairly uniform model of returning –1 when an error occurs, with the actual error value in the `errno` process variable. The threads API returns 0 on success but a positive value to indicate an error.

The `pthreads` API

While the `pthreads` API is comprehensive, it's quite easy to understand and use. This section explores the `pthreads` API, looking at the basics of thread creation through the specialized communication and synchronization methods that are available.

All multithreaded programs must make the `pthread` function prototypes and symbols available for use. This is accomplished by including the `pthread` standard header, as follows:

```
#include <pthread.h>
```

> NOTE
>
> *The examples that follow are written for brevity, and in some cases, return values are not checked. To avoid debugging surprises, you are strongly encouraged to check all system call return values and never assume that a function is successful.*

Thread Basics

All multithreaded applications must create threads and ultimately destroy them. This is provided in two functions by the `pthreads` API:

```
int pthread_create( pthread_t *thread,
                    pthread_attr_t *attr,
                void *(*start_routine)(void *), void *arg );
int pthread_exit( void *retval );
```

The `pthread_create` function permits the creation of a new thread, whereas `pthread_exit` allows a thread to terminate itself. You also have a function to permit one thread to terminate another, but that is investigated later.

To create a new thread, you call `pthread_create` and associate your `pthread_t` object with a function (`start_routine`). This function represents the top level code that is executed within the thread. You can optionally provide a set of attributes via `pthread_attr_t` (via `pthread_attr_init`). Finally, the fourth argument (`arg`) is an optional argument that is passed to the thread upon creation.

Now it's time to take a look at a short example of thread creation (see Listing 15.1). In the `main` function, you first create a `pthread_t` object at line 10. This object represents your new thread. You call `pthread_create` at line 12 and provide the `pthread_t` object (which is filled in by the `pthread_create` function) in addition to your function that contains the code for the thread (argument 3, `myThread`). A zero return indicates successful creation of the thread.

LISTING 15.1 Creating a Thread with `pthread_create` (on the CD-ROM at `./source/ch15/ptcreate.c`)

```
1:      #include <pthread.h>
2:      #include <stdlib.h>
3:      #include <stdio.h>
4:      #include <string.h>
5:      #include <errno.h>
6:
7:      int main()
8:      {
9:        int ret;
10:       pthread_t mythread;
11:
12:       ret = pthread_create( &mythread, NULL, myThread, NULL );
13:
14:       if (ret != 0) {
15:         printf( "Can't create pthread (%s)\n", strerror(
            errno ) );
16:         exit(-1);
17:       }
18:
19:       return 0;
20:     }
```

The `pthread_create` function returns zero if successful; otherwise, a nonzero value is returned. Now it's time to take a look at the thread function itself, which also demonstrates the `pthread_exit` function (see Listing 15.2). The thread simply emits a message to `stdout` that it ran and then terminated at line 6 with `pthread_exit`.

LISTING 15.2 Terminating a Thread with `pthread_exit` (on the CD-ROM at `./source/ch15/ptcreate.c`)

```
1:      void *myThread( void *arg )
2:      {
```

```
        printf("Thread ran!\n");

        /* Terminate the thread */
        pthread_exit( NULL );
      }
```

This thread didn't use the `void` pointer argument, but this could be used to provide the thread with a specific personality, passed in at creation (see the fourth argument of line 12 in Listing 15.1). The argument can represent a scalar value or a structure containing a variety of elements. The `exit` value presented to `pthread_exit` must not be of local scope; otherwise, it won't exist after the thread is destroyed. The `pthread_exit` function does not return.

NOTE

The startup cost for new threads is minimal in the new NPTL implementation, compared to the older LinuxThreads. In addition to significant improvements and optimizations in the NPTL, the allocation of thread memory structures is improved (thread data structures and thread local storage are now provided on the local thread stack).

Thread Management

Before digging into thread synchronization and coordination, it's time to look at a couple of miscellaneous thread functions that can be of use. The first is the `pthread_self` function, which can be used by a thread to retrieve its unique identifier. Recall in `pthread_create` that a `pthread_t` object reference is passed in as the first argument. This permits the thread creator to know the identifier for the thread just created. The thread itself can also retrieve this identifier by calling `pthread_self`.

```
pthread_t pthread_self( void );
```

Consider the updated thread function in Listing 15.3, which illustrates retrieving the `pthread_t` handle. At line 5, you call `pthread_self` to grab the handle and then emit it to `stdout` at line 7 (converting it to an `int`).

LISTING 15.3 Retrieving the `pthread_t` Handle with `pthread_self` (on the CD-ROM at `./source/ch15/ptcreate.c`)

```
1:      void *myThread( void *arg )
2:      {
3:        pthread_t pt;
4:
```

```
    pt = pthread_self();

    printf("Thread %x ran!\n", (int)pt );

    pthread_exit( NULL );
}
```

Most applications require some type of initialization, but with threaded applications, the job can be difficult. The `pthread_once` function allows a developer to create an initialization routine that is invoked for a multithreaded application only once (even though multiple threads might attempt to invoke it).

The `pthread_once` function requires two objects: a `pthread_once_t` object (that has been preinitialized with `pthread_once_init`) and an initialization function. Consider the partial example in Listing 15.4. The first thread to call `pthread_once` invokes the initialization function (`initialize_app`), but subsequent calls to `pthread_once` result in no calls to `initialize_app`.

LISTING 15.4 Providing a Single-Use Initialization Function with `pthread_once`

```
#include <pthread.h>

pthread_once_t my_init_mutex = pthread_once_init;

void initialize_app( void )
{
  /* Single-time init here */
}

void *myThread( void *arg )
{
  ...

  pthread_once( &my_init_mutex, initialize_app );

  ...
}
```

NOTE

The number of threads in LinuxThreads was a compile-time option (1000), whereas NPTL supports a dynamic number of threads. NPTL can support up to 2 billion threads on an IA-32 system [Drepper and Molnar03].

Thread Synchronization

The ability to synchronize threads is an important aspect of multithreaded application development. This chapter looks at a number of methods, but first you need to take a look at the most basic method, the ability for the creator thread to wait for the created thread to finish (otherwise known as a `join`). This activity is provided by the `pthread_join` API function. When called, the `pthread_join` call suspends the calling thread until a `join` is complete. When the `join` is done, the caller receives the joined thread's termination status as the return from `pthread_join`. The `pthread_join` function (somewhat equivalent to the `wait` function for processes) has the following prototype:

```
int pthread_join( pthread_t th, void **thread_return );
```

The `th` argument is the thread to which you want to join. This argument is returned from `pthread_create` or passed via the thread itself via `pthread_self`. The `thread_return` can be `NULL`, which means you do not capture the return status of the thread. Otherwise, the return value from the thread is stored in `thread_return`.

NOTE

A thread is automatically joinable when using the default attributes of `pthread_create`. *If the attribute for the thread is defined as detached, then the thread can't be joined (because it's detached from the creating thread).*

To join with a thread, you must have the thread's identifier, which is retrieved from the `pthread_create` function. Take a look at a complete example (see Listing 15.5).

In this example, you permit the creation of five distinct threads by calling `pthread_create` within a loop (lines 18–23) and storing the resulting thread identifiers in a `pthread_t` array (line 16). After the threads are created, you begin the `join` process, again in a loop (lines 25–32). The `pthread_join` returns zero on success, and upon success, the `status` variable is emitted (note that this value is returned at line 8 within the thread itself).

LISTING 15.5 Joining Threads with `pthread_join` (on the CD-ROM at `./source/ch15/ptjoin.c`)

```
1:      #include <pthread.h>
2:      #include <stdio.h>
3:
4:      void *myThread( void *arg )
5:      {
```

```
  printf( "Thread %d started\n", (int)arg );

  pthread_exit( arg );
}

#define MAX_THREADS    5

int main()
{
  int ret, i, status;
  pthread_t threadIds[MAX_THREADS];

  for (i = 0 ; i < MAX_THREADS ; i++) {
    ret = pthread_create( &threadIds[i], NULL, myThread,
    (void *)i );
    if (ret != 0) {
      printf( "Error creating thread %d\n",
      (int)threadIds[i] );
    }
  }

  for (i = 0 ; i < MAX_THREADS ; i++) {
    ret = pthread_join( threadIds[i], (void **)&status );
    if (ret != 0) {
      printf( "Error joining thread %d\n",
      (int)threadIds[i] );
    } else {
      printf( "Status = %d\n", status );
    }
  }

  return 0;
}
```

The `pthread_join` function suspends the caller until the requested thread has been joined. In many cases, you simply don't care about the thread after it's created. In these cases, you can identify this by detaching the thread. The creator or the thread itself can detach itself. You can also specify that the thread is detached when you create the thread (as part of the attributes). After a thread is detached, it can never be joined. The `pthread_detach` function has the following prototype:

```
int pthread_detach( pthread_t th );
```

Now take a look at the process of detaching the thread within the thread itself (see Listing 15.6). Recall that a thread can identify its own identifier by calling `thread_self`.

LISTING 15.6 Detaching a Thread from Within with `pthread_detach`

```
1:    void *myThread( void *arg )
2:    {
3:      printf( "Thread %d started\n", (int)arg );
4:
5:      pthread_detach( pthread_self() );
6:
7:      pthread_exit( arg );
8:    }
```

At line 5, you simply call `pthread_detach`, specifying the thread identifier by calling `pthread_self`. When this thread exits, all resources are immediately freed (as it's detached and will never be joined by another thread). The `pthread_detach` function returns zero on success, nonzero if an error occurs.

NOTE

GNU/Linux automatically places a newly created thread into the joinable state. This is not the case in other implementations, which can default to detached.

Thread Mutexes

A mutex is a variable that permits threads to implement critical sections. These sections enforce exclusive access to variables by threads, which if left unprotected result in data corruption. This topic is discussed in detail in Chapter 17, "Synchronization with Semaphores."

This section starts by reviewing the mutex API, and then illustrates the problem being solved. To create a mutex, you simply declare a variable that represents your mutex and initializes it with a special symbolic constant. The mutex is of type `pthread_mutex_t` and is demonstrated as follows:

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER
```

As shown here, the initialization makes this mutex a fast mutex. The mutex initializer can actually be of one of three types, as shown in Table 15.1.

TABLE 15.1 Mutex Initializers

Type	Description
PTHREAD_MUTEX_INITIALIZER	Fast mutex
PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP	Recursive mutex
PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP	Error-checking mutex

The recursive mutex is a special mutex that allows the mutex to be locked several times (without blocking), as long as it's locked by the same thread. Even though the mutex can be locked multiple times without blocking, the thread must unlock the mutex the same number of times that it was locked. The error-checking mutex can be used to help find errors when debugging. Note that the `_NP` suffix for recursive and error-checking mutexes indicates that they are not portable.

Now that you have a mutex, you can lock and unlock it to create your critical section. This is done with the `pthread_mutex_lock` and `pthread_mutex_unlock` API functions. Another function called `pthread_mutex_trylock` can be used to try to lock a mutex, but it won't block if the mutex is already locked. Finally, you can destroy an existing mutex using `pthread_mutex_destroy`. These have the prototype as follows:

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mutex_t *mutex );
int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

All functions return zero on success or a nonzero error code. All errors returned from `pthread_mutex_lock` and `pthread_mutex_unlock` are assertable (not recoverable). Therefore, you use the return of these functions to abort your program.

Locking a thread is the means by which you enter a critical section. After your mutex is locked, you can safely enter the section without having to worry about data corruption or multiple access. To exit your critical section, you unlock the semaphore and you're done. The following code snippet illustrates a simple critical section:

```
pthread_mutex_t cntr_mutex = PTHREAD_MUTEX_INITIALIZER;
...
assert( pthread_mutex_lock( &cntr_mutex ) == 0 );
/* Critical Section */
```

```
/* Increment protected counter */
counter++;
/* Critical Section */
  assert( pthread_mutex_unlock( &cntr_mutex ) == 0 );
```

NOTE

A critical section is a section of code that can be executed by at most one process at a time. The critical section exists to protect shared resources from multiple access.

The `pthread_mutex_trylock` operates under the assumption that if you can't lock your mutex, you should do something else instead of blocking on the `pthread_mutex_lock` call. This call is demonstrated as follows:

```
ret = pthread_mutex_trylock( &cntr_mutex );
if (ret == EBUSY) {
  /* Couldn't lock, do something else */
} else if (ret == EINVAL) {
  /* Critical error */
  assert(0);
} else {
  /* Critical Section */
  ret = thread_mutex_unlock( &cntr_mutex );
}
```

Finally, to destroy your mutex, you simply provide it to the `pthread_mutex_destroy` function. The `pthread_mutex_destroy` function succeeds only if no thread currently has the mutex locked. If the mutex is locked, the function fails and returns the `EBUSY` error code. The `pthread_mutex_destroy` call is demonstrated with the following snippet:

```
ret = pthread_mutex_destroy( &cntr_mutex );
if (ret == EBUSY) {
  /* Mutex is locked, can't destroy */
} else {
  /* Mutex was destroyed */
}
```

Now take a look at an example that ties these functions together to illustrate why mutexes are important in multithreaded applications. In this example, you build on the previous applications that provide a basic infrastructure for task creation and joining. Consider the example in Listing 15.7. At line 4, you create your mutex and initialize it as a fast mutex. In your thread, your job is to increment the `protVariable` counter some number of times. This occurs for each thread (here you create 10), so you need to protect the variable from multiple access. You place the variable increment within a critical section by first locking the mutex and then, after incrementing the protected variable, unlocking it. This ensures that each task has sole access to the resource when the increment is performed and protects it from corruption. Finally, at line 52, you destroy your mutex using the `pthread_mutex_destroy` API function.

LISTING 15.7 Protecting a Variable in a Critical Section with Mutexes (on the CD-ROM at `./source/ch15/ptmutex.c`)

```
1:        #include <pthread.h>
2:        #include <stdio.h>
3:
4:        pthread_mutex_t cntr_mutex = PTHREAD_MUTEX_INITIALIZER;
5:
6:        long protVariable = 0L;
7:
8:        void *myThread( void *arg )
9:        {
10:         int i, ret;
11:
12:         for (i = 0 ; i < 10000 ; i++) {
13:
14:           ret = pthread_mutex_lock( &cntr_mutex );
15:
16:           assert( ret == 0 );
17:
18:           protVariable++;
19:
20:           ret = pthread_mutex_unlock( &cntr_mutex );
21:
22:           assert( ret == 0 );
23:
24:         }
25:
26:         pthread_exit( NULL );
27:       }
28:
29:       #define MAX_THREADS     10
30:
31:       int main()
32:       {
33:         int ret, i;
```

```
        pthread_t threadIds[MAX_THREADS];

        for (i = 0 ; i < MAX_THREADS ; i++) {
          ret = pthread_create( &threadIds[i], NULL, myThread,
          NULL );
          if (ret != 0) {
            printf( "Error creating thread %d\n",
            (int)threadIds[i] );
          }
        }

        for (i = 0 ; i < MAX_THREADS ; i++) {
          ret = pthread_join( threadIds[i], NULL );
          if (ret != 0) {
            printf( "Error joining thread %d\n",
            (int)threadIds[i] );
          }
        }

        printf( "The protected variable value is %ld\n",
        protVariable );

        ret = pthread_mutex_destroy( &cntr_mutex );

        if (ret != 0) {
          printf( “Couldn’t destroy the mutex\n”);
        }

        return 0;
      }
```

When using mutexes, it’s important to minimize the amount of work done in the critical section to what really needs to be done. Because other threads block until a mutex is unlocked, minimizing the critical section time can lead to better performance.

Thread Condition Variables

Now that you have mutexes out of the way, you can explore condition variables. A condition variable is a special thread construct that allows a thread to wake up another thread based upon a condition. Whereas mutexes provide a simple form of synchronization (based upon the lock status of the mutex), condition variables are a means for one thread to wait for an event and another to signal it that the event has occurred. An event can mean anything here. A thread blocks on a mutex but can wait on any condition variable. Think of them as wait queues, which is exactly what the implementation does in GNU/Linux.

Consider this problem of a thread awaiting a particular condition being met. If you use only mutexes, the thread has to poll to acquire the mutex, check the condition, and then release the mutex if no work is found to do (the condition isn't met). That kind of busy looping can lead to poorly performing applications and needs to be avoided.

The `pthreads` API provides a number of functions supporting condition variables. These functions provide condition variable creation, waiting, signaling, and destruction. The condition variable API functions are presented as follows:

```
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
int pthread_cond_timedwait( pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec *abstime );
int pthread_cond_signal( pthread_cond_t *cond );
int pthread_cond_broadcast( pthread_cond_t *cond );
int pthread_cond_destroy( pthread_cond_t *cond );
```

To create a condition variable, you simply create a variable of type `pthread_cond_t`. You initialize this by setting it to `PTHREAD_COND_INITIALIZER` (similar to mutex creation and initialization). This is demonstrated as follows:

```
pthread_cond_t recoveryCond = PTHREAD_COND_INITIALIZER;
```

Condition variables require the existence of a mutex that is associated with them, which you create as you learned previously:

```
pthread_mutex_t recoveryMutex = PTHREAD_MUTEX_INITIALIZER;
```

Now take a look at a thread awaiting a condition. In this example, say you have a thread whose job is to warn of overload conditions. Work comes in on a queue, with an accompanying counter identifying the amount of work to do. When the amount of work exceeds a certain value (`MAX_NORMAL_WORKLOAD`), then your thread wakes up and performs a recovery. Your fault thread for synchronizing with the alert thread is illustrated as follows:

```
/* Fault Recovery Thread Loop */
while ( 1 ) {
  assert( pthread_mutex_lock( &recoveryMutex ) == 0);
  while (workload < MAX_NORMAL_WORKLOAD) {
    pthread_cond_wait( &recoveryCond, &recoveryMutex );
  }
  /*————————*/
  /* Recovery Code. */
  /*————————*/
  assert( pthread_mutex_unlock( &recoveryMutex ) == 0);
}
```

This is the standard pattern when dealing with condition variables. You start by locking the mutex, entering pthread_cond_wait, and upon waking up from your condition, unlocking the mutex. The mutex must be locked first because upon entry to pthread_cond_wait, the mutex is automatically unlocked. When you return from pthread_cond_wait, the mutex has been reacquired, meaning that you need to unlock it afterward. The mutex is necessary here to handle race conditions that exist in this call sequence. To ensure that your condition is met, you loop around the pthread_cond_wait, and if the condition is not satisfied (in this case, your workload is normal), then you reenter the pthread_cond_wait call. Note that because the mutex is locked upon return from pthread_cond_wait, you don't need to call pthread_mutex_lock here.

Now take a look at the signal code. This is considerably simpler than that code necessary to wait for the condition. Two possibilities exist for signaling: sending a single signal or broadcasting to all waiting threads.

The first case is signaling one thread. In either case, you first lock the mutex before calling the signal function and then unlock when you're done. To signal one thread, you call the pthread_cond_signal function, as follows:

```
pthread_mutex_lock( &recoveryMutex );
pthread_cond_signal( &recoveryCond );
pthread_mutex_unlock( &recovery_Mutex );
```

After the mutex is unlocked, exactly one thread is signaled and allowed to execute. Each function returns zero on success or an error code. If your architecture supports multiple threads for recovery, you can instead use the pthread_cond_broadcast. This function wakes up all threads currently awaiting the condition. This is demonstrated as follows:

```
pthread_mutex_lock( &recoveryMutex );
pthread_cond_broadcast( &recoveryCond );
pthread_mutex_unlock( &recovery_Mutex );
```

After the mutex is unlocked, the series of threads is then permitted to perform recovery (though one by one because they're dependent upon the mutex).

The pthreads API also supports a version of timed-wait for a condition variable. This function, pthread_cond_timedwait, allows the caller to specify an absolute time representing when to give up and return to the caller. The return value is ETIMEDOUT, to indicate that the function returned because of a timeout rather than because of a successful return. The following code snippet illustrates its use:

```
struct timeval currentTime;
struct timespec  expireTime;
int ret;
...
assert( pthread_mutex_lock( &recoveryMutex ) == 0);
gettimeofday( &currentTime );
expireTime.tv_sec = currentTime.tv_sec + 1;
expireTime.tv_nsec = currentTime.tv_usec * 1000;
ret = 0;
while ((workload < MAX_NORMAL_WORKLOAD) && (ret != ETIMEDOUT) {
  ret = pthread_cond_timedwait( &recoveryCond, &recoveryMutex,
                &expireTime );
}
if (ret == ETIMEDOUT) {
  /* Timeout – perform timeout processing */
} else {
  /* Condition met – perform condition recovery processing */
}
assert( pthread_mutex_unlock( &recoveryMutex ) == 0);
```

The first item to note is the generation of a timeout. You use the gettimeofday function to get the current time and then add one second to it in the timespec structure. This is passed to pthread_cond_timedwait to identify the time at which you desire a timeout if the condition has not been met. In this case, which is very similar to the standard pthread_cond_wait example, you check in your loop that the pthread_cond_timedwait function has not returned ETIMEDOUT. If it has, you exit your loop and then check again to perform timeout processing. Otherwise, you perform your standard condition processing (recovery for this example) and then reacquire the mutex.

The final function to note here is pthread_cond_destroy. You simply pass the condition variable to the function, as follows:

```
pthread_mutex_destroy( &recoveryCond );
```

It's important to note that in the GNU/Linux implementation no resources are actually attached to the condition variable, so this function simply checks to see if any threads are currently pending on the condition variable.

Now it's time to look at a complete example that brings together all of the elements just discussed for condition variables. This example illustrates condition variables in the context of producers and consumers. You create a producer thread that creates work and then *N* consumer threads that operate on the (simulated) work.

The first listing (Listing 15.8) shows the `main` program. This listing is similar to the previous examples of creating and then joining threads, with a few changes. You create two types of threads in this listing. At lines 18–21, you create a number of consumer threads, and at line 24, you create a single producer thread. You will take a look at these shortly. After creation of the last thread, you join the producer thread (resulting in a suspend of the main application until it has completed). You then wait for the work to complete (as identified by a simple counter, `workCount`). You want to allow the consumer threads to complete their work, so you wait until this variable is zero, indicating that all work is consumed.

The block of code at lines 33–36 shows joins for the consumer threads, with one interesting change. In this example, the consumer threads never quit, so you cancel them here using the `pthread_cancel` function. This function has the following prototype:

```
int pthread_cancel( pthread_t thread );
```

This permits you to terminate another thread when you're done with it. In this example, you have produced the work that you need the consumers to work on, so you cancel each thread in turn (line 34). Finally, you destroy your condition variable and mutex at lines 37 and 38, respectively.

LISTING 15.8 Producer/Consumer Example Initialization and `main` (on the CD-ROM at `./source/ch15/ptcond.c`)

```
1:     #include <pthread.h>
2:     #include <stdio.h>
3:
4:     pthread_mutex_t cond_mutex = PTHREAD_MUTEX_INITIALIZER;
5:     pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
6:
7:     int workCount = 0;
8:
9:     #define MAX_CONSUMERS   10
10:
```

```
int main()
{
  int i;
  pthread_t consumers[MAX_CONSUMERS];
  pthread_t producer;

  /* Spawn the consumer thread */
  for ( i = 0 ; i < MAX_CONSUMERS ; i++ ) {
    pthread_create( &consumers[i], NULL,
                        consumerThread, NULL );
  }

  /* Spawn the single producer thread */
  pthread_create( &producer, NULL,
                        producerThread, NULL );

  /* Wait for the producer thread */
  pthread_join( producer, NULL );

  while ((workCount > 0));

  /* Cancel and join the consumer threads */
  for ( i = 0 ; i < MAX_CONSUMERS ; i++ ) {
    pthread_cancel( consumers[i] );
  }

  pthread_mutex_destroy( &cond_mutex );
  pthread_cond_destroy( &condition );

  return 0;
}
```

Next, you can take a look at the producer thread function (Listing 15.9). The purpose of the producer thread is to produce work, simulated by incrementing the `workCount` variable. A nonzero `workCount` indicates that work is available to do. You loop for a number of times to create work, as is shown at lines 8–22. As shown in the condition variable sample, you first lock your mutex at line 10 and then create work to do (increment `workCount`). You then notify the awaiting consumer (worker) threads at line 14 using the `pthread_cond_broadcast` function. This notifies any awaiting consumer threads that work is now available to do. Next, at line 15, you unlock the mutex, allowing the consumer threads to lock the mutex and perform their work.

At lines 20–22, you simply do some busy work to allow the kernel to schedule another task (thereby avoiding synchronous behavior, for illustration purposes).

When all of the work has been produced, you permit the producer thread to exit (which is joined in the `main` function at line 28 of Listing 15.8).

LISTING 15.9 Producer Thread Example for Condition Variables (on the CD-ROM at `./source/ch15/ptcond.c`)

```
1:    void *producerThread( void *arg )
2:    {
3:      int i, j, ret;
4:      double result=0.0;
5:
6:      printf("Producer started\n");
7:
8:      for ( i = 0 ; i < 30 ; i++ ) {
9:
10:       ret = pthread_mutex_lock( &cond_mutex );
11:       if (ret == 0) {
12:         printf( "Producer: Creating work (%d)\n", workCount );
13:         workCount++;
14:         pthread_cond_broadcast( &condition );
15:         pthread_mutex_unlock( &cond_mutex );
16:       } else {
17:         assert(0);
18:       }
19:
20:       for ( j = 0 ; j < 60000 ; j++ ) {
21:         result = result + (double)random();
22:       }
23:
24:     }
25:
26:     printf("Producer finished\n");
27:
28:     pthread_exit( NULL );
29:   }
```

Now it's time to look at the consumer thread (see Listing 15.10). Your first task is to detach yourself (line 5), because you won't ever join with the creating thread. Then you go into your work loop (lines 9–22) to process the workload. You first lock the condition mutex at line 11 and then wait for the condition to occur at line 12. You then check to make sure that the condition is true (that work exists to do) at line 14. Note that because you're broadcasting to threads, you might not have work to do for every thread, so you test before you assume that work is available.

After you've completed your work (in this case, simply decrementing the work count at line 15), you release the mutex at line 19 and wait again for work at line 11. Note that because you cancel your thread, you never see the `printf` at line 23, nor do you exit the thread at line 25. The `pthread_cancel` function terminates the thread so that the thread does not terminate normally.

LISTING 15.10 Consumer Thread Example for Condition Variables (on the CD-ROM at `./source/ch15/ptcond.c`)

```
1:    void *consumerThread( void *arg )
2:    {
3:      int ret;
4:
5:      pthread_detach( pthread_self() );
6:
7:      printf( “Consumer %x: Started\n”, pthread_self() );
8:
9:      while( 1 ) {
10:
11:       assert( pthread_mutex_lock( &cond_mutex ) == 0);
12:       assert( pthread_cond_wait( &condition, &cond_mutex )
          == 0 );
13:
14:       if (workCount) {
15:         workCount–;
16:         printf( “Consumer %x: Performed work (%d)\n”,
17:                   pthread_self(), workCount );
18:       }
19:       assert( pthread_mutex_unlock( &cond_mutex ) == 0);
20:
21:     }
22:
23:     printf( “Consumer %x: Finished\n”, pthread_self() );
24:
25:     pthread_exit( NULL );
26:   }
```

Now take a look at this application in action. For brevity, this example shows only the first 30 lines emitted, but this gives you a good indication of how the application behaves (see Listing 15.11). You can see the consumer threads starting up, the producer starting, and then work being created and consumed in turn.

LISTING 15.11 Application Output for Condition Variable Application

```
$ ./ptcond
Consumer 4082cd40: Started
Consumer 4102ccc0: Started
Consumer 4182cc40: Started
Consumer 42932bc0: Started
Consumer 43132b40: Started
Consumer 43932ac0: Started
Consumer 44132a40: Started
Consumer 449329c0: Started
Consumer 45132940: Started
Consumer 459328c0: Started
Producer started
Producer: Creating work (0)
Producer: Creating work (1)
Consumer 4082cd40: Performed work (1)
Consumer 4102ccc0: Performed work (0)
Producer: Creating work (0)
Consumer 4082cd40: Performed work (0)
Producer: Creating work (0)
Producer: Creating work (1)
Producer: Creating work (2)
Producer: Creating work (3)
Producer: Creating work (4)
Producer: Creating work (5)
Consumer 4082cd40: Performed work (5)
Consumer 4102ccc0: Performed work (4)
Consumer 4182cc40: Performed work (3)
Consumer 42932bc0: Performed work (2)
Consumer 43132b40: Performed work (1)
Consumer 43932ac0: Performed work (0)
Producer: Creating work (0)
```

NOTE

The design of multithreaded applications follows a small number of patterns (or models). The master/servant model is common where a single master doles out work to a collection of servants. The pipeline model splits work up into stages where one or more threads make up each of the work phases.

BUILDING THREADED APPLICATIONS

Building pthread-based applications is very simple. All that's necessary is to specify the `pthreads` library during compilation as follows:

```
gcc -pthread threadapp.c -o threadapp –lpthread
```

This links your application with the `pthread` library, making the `pthread` functions available for use. Note also that you specify the `-pthread` option, which adds support for multithreading to the application (such as re-entrancy). The option also ensures that certain global system variables (such as `errno`) are provided on a per-thread basis.

One topic that's important to discuss in multithreaded applications is that of re-entrancy. Consider two threads, each of which uses the `strtok` function. This function uses an internal buffer for token processing of a string. This internal buffer can be used by only one user at a time, which is fine in the process world (forked processes), but in the thread world runs into problems. If each thread attempts to call `strtok`, then the internal buffer is corrupted, leading to undesirable (and unpredictable) behavior. To fix this, rather than using an internal buffer, you can use a thread-supplied buffer instead. This is exactly what happens with the thread-safe version of `strtok`, called `strtok_r`. The suffix `_r` indicates that the function is thread-safe.

Summary

Multithreaded application development is a powerful model for the development of high-performance software systems. GNU/Linux provides the POSIX `pthreads` API for a standard and portable programming model. This chapter explored the standard thread creation, termination, and synchronization functions. This includes the basic synchronization using a join, but also more advanced coordination using mutexes and condition variables. Finally, building `pthread` applications was investigated, along with some of the pitfalls (such as re-entrancy) and how to deal with them. The GNU/Linux 2.6 kernel (using NPTL) provides a closer POSIX implementation and more efficient IPC and kernel support than the prior Linux-Threads version provided.

References

[Drepper and Molnar03] Drepper, Ulrich and Molnar, Ingo. (2003) *The Native POSIX Thread Library for Linux*. Red Hat, Inc.

API Summary

```
#include <pthread.h>
int pthread_create( pthread_t *thread,
                    pthread_attr_t *attr,
                    void *(*start_routine)(void *), void *arg );
int pthread_exit( void *retval );
pthread_t pthread_self( void );
int pthread_join( pthread_t th, void **thread_return );
int pthread_detach( pthread_t th );
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mutex_t *mutex );
int pthread_mutex_destroy( pthread_mutex_t *mutex );
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
int pthread_cond_timedwait( pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec *abstime );
int pthread_cond_signal( pthread_cond_t *cond );
int pthread_cond_broadcast( pthread_cond_t *cond );
int pthread_cancel( pthread_t thread );
```

16 IPC with Message Queues

In This Chapter

- Introduction to Message Queues
- Creating and Configuring Message Queues
- Creating Messages Suitable for Message Queues
- Sending and Receiving Messages
- Adjusting Message Queue Behavior
- The `ipcs` Utility

Introduction

The topic of interprocess communication is an important one because it allows you the ability to build systems out of numerous communicating asynchronous processes. This is beneficial because you can naturally segment the functionality of a large system into a number of distinct elements. Because GNU/Linux processes utilize independent memory spaces, a function in one process cannot call another in a different process. Message queues provide one means to permit communication and coordination between processes. This chapter reviews the message queue model (which conforms to the SystemV UNIX model), as well as explores some sample code that utilizes the message queue API.

QUICK OVERVIEW OF MESSAGE QUEUES

This chapter begins by taking a whirlwind tour of the POSIX-compliant message queue API. You will take a look at code examples that illustrate creating a message queue, configuring its size, sending and receiving a message, and then removing the message queue. After you have had a taste of the message queue API, you can dive in deeper in the following sections.

Using the message queue API requires that the function prototypes and symbols be available to the application. This is done by including the `msg.h` header file as follows:

```
#include <sys/msg.h>
```

You first introduce a common header file that defines some common information needed for the writer and reader of the message (see Listing 16.1). You define your system-wide queue ID (111) at line 3. This isn't the best way to define the queue, but later on you will see a way to define a unique system ID. Lines 5–10 define your message type, with the required long type at the head of the structure (line 6).

LISTING 16.1 Common Header File Used by the Sample Applications (on the CD-ROM at `./source/ch16/common.h`)

```
1:      #define MAX_LINE        80
2:
3:      #define MY_MQ_ID        111
4:
5:      typedef struct {
6:              long  type;             // Msg Type (> 0)
7:              float fval;             // User Message
8:              unsigned int uival;     // User Message
9:              char strval[MAX_LINE+1]; // User Message
10:     } MY_TYPE_T;
```

CREATING A MESSAGE QUEUE

To create a message queue, you use the `msgget` API function. This function takes a message queue ID (a unique identifier, or key, within a given host) and another argument identifying the message flags. The flags in the queue create example (see Listing 16.2) specify that a queue is to be created (`IPC_CREAT`) as well as the access permissions of the message queue (read/write permission for system, user, and group).

NOTE

The result of the `msgget` *function is a handle, which is similar to a file descriptor, pointing to the message queue with the particular ID.*

LISTING 16.2 Creating a Message Queue with `msgget` (on the CD-ROM at `./source/ch16/mqcreate.c`)

```
1:        #include <stdio.h>
2:        #include <sys/msg.h>
3:        #include "common.h"
4:
5:        int main()
6:        {
7:          int msgid;
8:
9:          /* Create the message queue with the id MY_MQ_ID */
10:         msgid = msgget( MY_MQ_ID, 0666 | IPC_CREAT );
11:
12:         if (msgid >= 0) {
13:
14:           printf( "Created a Message Queue %d\n", msgid );
15:
16:         }
17:
18:         return 0;
19:       }
```

Upon creating the message queue at line 10 (in Listing 16.2), you get a return integer that represents a handle for the message queue. This message queue ID can be used in subsequent message queue calls to send or receive messages.

Configuring a Message Queue

When you create a message queue, some of the details of the process that created the queue are automatically stored with it (for permissions) as well as a default queue size in bytes (16 KB). You can adjust this size using the `msgctl` API function. Listing 16.3 illustrates reading the defaults for the message queue, adjusting the queue size, and then configuring the queue with the new set.

LISTING 16.3 Configuring a Message Queue with `msgctl` (on the CD-ROM at `./source/ch16/mqconf.c`)

```
#include <stdio.h>
#include <sys/msg.h>
#include "common.h"

int main()
{
```

```
7:         int msgid, ret;
8:         struct msqid_ds buf;
9:
10:        /* Get the message queue for the id MY_MQ_ID */
11:        msgid = msgget( MY_MQ_ID, 0 );
12:
13:        /* Check successful completion of msgget */
14:        if (msgid >= 0) {
15:
16:          ret = msgctl( msgid, IPC_STAT, &buf );
17:
18:          buf.msg_qbytes = 4096;
19:
20:          ret = msgctl( msgid, IPC_SET, &buf );
21:
22:          if (ret == 0) {
23:
24:            printf( "Size successfully changed for queue
               %d.\n", msgid );
25:
26:          }
27:
28:        }
29:
30:        return 0;
31:      }
```

First, at line 11, you get the message queue ID using `msgget`. Note that the second argument here is zero because you're not creating the message queue, just retrieving its ID. You use this at line 16 to get the current queue data structure using the `IPC_STAT` command and your local buffer (for which the function fills in the defaults). You adjust the queue size at line 18 (by modifying the `msg_qbytes` field of the structure) and then write it back at line 20 using the `msgctl` API function with the `IPC_SET` command. You can also modify the user or group ID of the message queue or its mode. This chapter discusses these capabilities in more detail later.

Writing a Message to a Message Queue

Now take a look at actually sending a message through a message queue. A message within the context of a message queue has only one constraint. The object that's being sent must include a long variable at its head that defines the message type. This is discussed more later in the chapter, but it's simply a way to differentiate messages that have been loaded onto a queue (and also how those messages can be read from the queue). The general structure for a message is as follows:

```
typedef struct {
  long type;
  char message[80];
} MSG_TYPE_T;
```

In this example (`MSG_TYPE_T`), you have your required `long` at the head of the message, followed by the user-defined message (in this case, a string of 80 characters).

To send a message to a message queue (see Listing 16.4), you use the `msgsnd` API function. Following a similar pattern to the previous examples, you first identify the message queue ID using the `msgget` API function (line 11). After this is known, you can send a message to it. Next, you initialize your message at lines 16–19. This includes specifying the mandatory type (must be greater than zero), a floating-point value (`fval`) and unsigned int value (`uival`), and a character string (`strval`). To send this message, you call the `msgsnd` API function. The arguments for this function are the message queue ID (`qid`), your message (a reference to `myObject`), the size of the message you're sending (the size of `MY_TYPE_T`), and finally a set of message flags (for now, `0`, but you'll investigate more later in the chapter).

LISTING 16.4 Sending a Message with `msgsnd` (on the CD-ROM at `./source/ch16/mqsend.c`)

```
1:    #include <sys/msg.h>
2:    #include <stdio.h>
3:    #include "common.h"
4:
5:    int main()
6:    {
7:      MY_TYPE_T myObject;
8:      int qid, ret;
9:
10:     /* Get the queue ID for the existing queue */
11:     qid = msgget( MY_MQ_ID, 0 );
12:
13:     if (qid >= 0) {
14:
15:       /* Create our message with a message queue type of 1 */
16:       myObject.type = 1L;
17:       myObject.fval = 128.256;
18:       myObject.uival = 512;
```

```
        strncpy( myObject.strval, "This is a test.\n",
        MAX_LINE );

        /* Send the message to the queue defined by the queue
        ID */
        ret = msgsnd( qid, (struct msgbuf *)&myObject,
                      sizeof(MY_TYPE_T), 0 );

        if (ret != -1) {

          printf( "Message successfully sent to queue %d\n",
          qid );

        }

      }

      return 0;
    }
```

That's it! This message is now held in the message queue, and at any point in the future, it can be read (and consumed) by the same or a different process.

Reading a Message from a Message Queue

Now that you have a message in your message queue, you can look at reading that message and displaying its contents (see Listing 16.5). You retrieve the ID of the message queue using `msgget` at line 12 and then use this as the target queue from which to read using the `msgrcv` API function at lines 16–17. The arguments to `msgrcv` are first the message queue ID (`qid`), the message buffer into which your message is to be copied (`myObject`), the size of the object (`sizeof(MY_TYPE_T)`), the message type that you want to read (`1`), and the message flags (`0`). Note that when you sent your message (in Listing 16.4), you specified our message type as `1`. You use this same value here to read the message from the queue. Had you used another value, the message would not have been read. More on this subject in the "`msgrcv`" section later in this chapter.

LISTING 16.5 Reading a Message with `msgrcv` (on the CD-ROM at `./source/ch16/mqrecv.c`)

```
1:     #include <sys/msg.h>
2:     #include <stdio.h>
```

```
#include "common.h"

int main()
{
  MY_TYPE_T myObject;
  int qid, ret;

  qid = msgget( MY_MQ_ID, 0 );

  if (qid >= 0) {

    ret = msgrcv( qid, (struct msgbuf *)&myObject,
                    sizeof(MY_TYPE_T), 1, 0 );

    if (ret != -1) {

      printf( "Message Type: %ld\n", myObject.type );
      printf( "Float Value:  %f\n",  myObject.fval );
      printf( "Uint Value:   %d\n",  myObject.uival );
      printf( "String Value: %s\n",  myObject.strval );

    }

  }

  return 0;
}
```

The final step in your application in Listing 16.5 is to emit the message read from the message queue. You use your object type to access the fields in the structure and simply emit them with `printf`.

Removing a Message Queue

As a final step, take a look at how you can remove a message queue (and any messages that might be held on it). You use the `msgctl` API function for this purpose with the command of `IPC_RMID`. This is illustrated in Listing 16.6.

LISTING 16.6 Removing a Message Queue with `msgctl` (on the CD-ROM at `./source/ch16/mqdel.c`)

```
#include <stdio.h>
#include <sys/msg.h>
#include "common.h"
```

```
int main()
{
  int   msgid, ret;

  msgid = msgget( MY_MQ_ID, 0 );

  if (msgid >= 0) {

    /* Remove the message queue */
    ret = msgctl( msgid, IPC_RMID, NULL );

    if (ret != -1) {

      printf( "Queue %d successfully removed.\n", msgid );

    }

  }

  return 0;
}
```

In Listing 16.6, you first identify the message queue ID using `msgget` and then use this with `msgctl` to remove the message queue. Any messages that happen to be on the message queue when `msgctl` is called are immediately removed.

That does it for our whirlwind tour. The next section digs deeper into the message queue API and looks at some of the behaviors of the commands that weren't covered already.

THE MESSAGE QUEUE API

Now it's time to dig into the message queue API and investigate each of the functions in more detail. For a quick review, Table 16.1 provides the API functions and their purposes.

Figure 16.1 graphically illustrates the message queue API functions and their relationship in the process.

The next sections address these functions in detail, identifying each of the uses with descriptive examples.

TABLE 16.1 Message Queue API Functions and Uses

API Function	Uses
msgget	Create a new message queue.
	Get a message queue ID.
msgsnd	Send a message to a message queue.
msgrcv	Receive a message from a message queue.
msgctl	Get the info about a message queue.
	Set the info for a message queue.
	Remove a message queue.

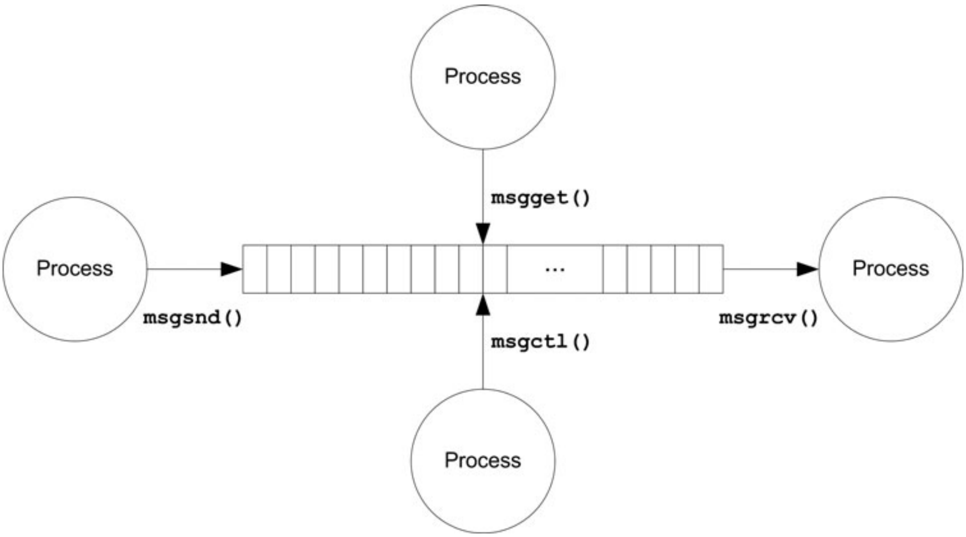

FIGURE 16.1 Message queue API functions.

msgget

The `msgget` API function serves two basic roles: to create a message queue or to get the identifier of a message queue that already exists. The result of the `msgget` function (unless an error occurs) is the message queue identifier (used by all other message queue API functions). The prototype for the `msgget` function is defined as follows:

```
int msgget( key_t key, int msgflag );
```

The key argument defines a system-wide identifier that uniquely identifies a message queue. key must be a nonzero value or the special symbol IPC_PRIVATE. The IPC_PRIVATE variable simply tells the msgget function that no key is provided and to simply make one up. The problem with this is that no other process can then find the message queue, but for local message queues (private queues), this method works fine.

The msgflag argument allows the user to specify two distinct parameters: a command and an optional set of access permissions. Permissions replicate those found as modes for the file creation functions (see Table 16.2). The command can take three forms. The first is simply IPC_CREAT, which instructs msgget to create a new message queue (or return the ID for the queue if it already exists). The second includes two commands (IPC_CREAT | IPC_EXCL), which request that the message queue be created, but if it already exists, the API function should fail and return an error response (EEXIST). The third possible command argument is simply 0. This form tells msgget that the message queue identifier for an existing queue is being requested.

TABLE 16.2 Message Queue Permissions for the msgget msgflag Argument

Symbol	Value	Meaning
S_IRUSR	0400	User has read permission.
S_IWUSR	0200	User has write permission.
S_IRGRP	0040	Group has read permission.
S_IWGRP	0020	Group has write permission.
S_IROTH	0004	Other has read permission.
S_IWOTH	0002	Other has write permission.

Now it's time to take a look at a few examples of the msgget function to create message queues or access existing ones. Assume in the following code snippets that msgid is an int value (int msgid). You can start by creating a private queue (no key is provided).

```
msgid = msgget( IPC_PRIVATE, IPC_CREAT | 0666 );
```

If the msgget API function fails, -1 is returned with the actual error value provided within the process's errno variable.

Now say that you want to create a message queue with a key value of `0x111`. You also want to know if the queue already exists, so you use the `IPC_EXCL` in this example:

```
// Create a new message queue
msgid = msgget( 0x111, IPC_CREAT | IPC_EXCL | 0666 );
 if (msgid == -1) {
   printf("Queue already exists...\n");
 } else {
   printf("Queue created...\n");
 }
```

An interesting question you've probably asked yourself now is how can you coordinate the creation of queues using IDs that might not be unique? What happens if someone already used the `0x111` key? Luckily, you have a way to create keys in a system-wide fashion that ensures uniqueness. The `ftok` system function provides the means to create system-wide unique keys using a file in the filesystem and a number. As the file (and its path) is by default unique in the filesystem, a unique key can be created easily. Take a look at an example of using `ftok` to create a unique key. Assume that the file with path `/home/mtj/queues/myqueue` exists.

```
key_t myKey;
int   msgid;
// Create a key based upon the defined path and number
myKey = ftok( "/home/mtj/queues/myqueue", 0 );
msgid = msgget( myKey, IPC_CREAT | 0666 );
```

This creates a key for this path and number. Each time `ftok` is called with this path and number, the same key is generated. Therefore, it provides a useful way to generate a key based upon a file in the filesystem.

One last example is getting the message queue ID of an existing message queue. The only difference in this example is that you provide no command, only the key:

```
msgid = msgget( 0x111, 0 );
if (msgid == -1) {
  printf("Queue doesn't exist...\n");
}
```

The `msgflags` (second argument to `msgget`) is zero in this case, which indicates to this API function that an existing message queue is being sought.

One final note on message queues relates to the default settings that are given to a message queue when it is created. The configuration of the message queue is noted in the parameters shown in Table 16.3. Note that you have no way to change these defaults within `msgget`. In the next section, you take look at some of the parameters that can be changed and their effects.

TABLE 16.3 Message Queue Configuration and Defaults in `msgget`

Parameter	Default Value
`msg_perm.cuid`	Effective user ID of the calling process (creator)
`msg_perm.uid`	Effective user ID of the calling process (owner)
`msg_perm.cgid`	Effective group ID of the calling process (creator)
`msg_perm.gid`	Effective group ID of the calling process (owner)
`msg_perm.mode`	Permissions (lower 9 bits of `msgflag`)
`msg_qnum`	0 (Number of messages in the queue)
`msg_lspid`	0 (Process ID of last `msgsnd`)
`msg_lrpid`	0 (Process ID of last `msgrcv`)
`msg_stime`	0 (last `msgsnd` time)
`msg_rtime`	0 (Last `msgrcv` time)
`msg_ctime`	Current time (last change time)
`msg_qbytes`	Queue size in bytes (system limit)—(16 KB)

The user can override the `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes` directly. More on this topic in the next section.

msgctl

The `msgctl` API function provides three distinct features for message queues. The first is the ability to read the current set of message queue defaults (via the `IPC_STAT` command). The second is the ability to modify a subset of the defaults (via `IPC_SET`). Finally, the ability to remove a message queue is provided (via `IPC_RMID`). The `msgctl` prototype function is defined as follows:

```
#include <sys/msg.h>
int msgctl( int msgid, int cmd, struct msqid_ds *buf );
```

You can start by looking at `msgctl` as a means to remove a message queue from the system. This is the simplest use of `msgctl` and can be demonstrated very easily. To remove a message queue, you need only the message queue identifier that is returned by `msgctl`.

NOTE

Whereas a system-wide unique key is required to create a message queue, only the message queue ID (returned from `msgget`*) is required to configure a queue, send a message from a queue, receive a message from a queue, or remove a queue.*

Now take a look at an example of message queue removal using `msgctl`. Whenever the shared resource is no longer needed, the application should remove it. You first get the message queue identifier using `msgget` and then use this ID in your call to `msgctl`.

```
int msgid, ret;
...
msgid = msgget( QUEUE_KEY, 0 );
if (msgid != -1) {
  ret = msgctl( msgid, IPC_RMID, NULL );
      if (ret == 0) {
    // queue was successfully removed.
  }
}
```

If any processes are currently blocked on a `msgsnd` or `msgrcv` API function, those functions return with an error (`-1`) with the `errno` process variable set to `EIDRM`. The process performing the `IPC_RMID` must have adequate permissions to remove the message queue. If permissions do not allow the removal, an error return is generated with an `errno` variable set to `EPERM`.

Now take a look at `IPC_STAT` (read configuration) and `IPC_SET` (write configuration) commands together for `msgctl`. In the previous section, you identified the range of parameters that make up the configuration and status parameters. Now it's time to look at which of the parameters can be directly manipulated or used by the application developer. Table 16.4 lists the parameters that can be updated after a message queue has been created.

Changing these parameters is a very simple process. The process should be that the application first reads the current set of parameters (via `IPC_STAT`) and then modifies the parameters of interest before writing them back out (via `IPC_SET`). See Listing 16.7 for an illustration of this process.

TABLE 16.4 Message Queue Parameters That Can Be Updated

Parameter	Description
`msg_perm.uid`	Message queue user owner
`msg_perm.gid`	Message queue group owner
`msg_perm.mode`	Permissions (see Table 16.2)
`msg_qbytes`	Size of message queue in bytes

LISTING 16.7 Setting All Possible Options in `msgctl` (on the CD-ROM at `./source/ch16/mqrdset.c`)

```
#include <stdio.h>
#include <sys/msg.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include "common.h"

int main()
{
  int msgid, ret;
  struct msqid_ds buf;

  /* Get the message queue for the id MY_MQ_ID */
  msgid = msgget( MY_MQ_ID, 0 );

  /* Check successful completion of msgget */
  if (msgid >= 0) {

    ret = msgctl( msgid, IPC_STAT, &buf );

    buf.msg_perm.uid = geteuid();
    buf.msg_perm.gid = getegid();
    buf.msg_perm.mode = 0644;
    buf.msg_qbytes = 4096;

    ret = msgctl( msgid, IPC_SET, &buf );

    if (ret == 0) {

```

```
30:             printf( "Parameters successfully changed.\n");
31:
32:           } else {
33:
34:             printf( "Error %d\n", errno );
35:
36:           }
37:
38:         }
39:
40:         return 0;
41:       }
```

At line 14, you get your message queue identifier, and then you use this at line 19 to retrieve the current set of parameters. At line 21, you set the `msg_perm.uid` (effective user ID) with the current effective user ID using the `geteuid()` function. Similarly, you set the `msg_perm.gid` (effective group ID) using the `getegid()` function at line 22. At line 23 you set the mode, and at line 24 you set the maximum queue size (in bytes). In this case you set it to 4 KB. You now take this structure and set the parameters for the current message queue using the `msgctl` API function. This is done with the `IPC_SET` command in `msgctl`.

NOTE

When setting the `msg_perm.mode` *(permissions), you need to know that this is traditionally defined as an octal value. Note at line 23 of Listing 16.7 that a leading zero is shown, indicating that the value is octal. If, for example, a decimal value of 666 were provided instead of octal 0666, permissions would be invalid, and therefore undesirable behavior would result. For this reason, it can be beneficial to use the symbols as shown in Table 16.2.*

You can also use the `msgctl` API function to identify certain message queue-specific parameters, such as the number of messages currently on the message queue. Listing 16.8 illustrates the collection and printing of the accessible parameters.

LISTING 16.8 Reading Current Message Queue Settings (on the CD-ROM at `./source/ch16/mqstats.c`)

```
#include <stdio.h>
#include <sys/msg.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include "common.h"

```

```
int main()
{
  int msgid, ret;
  struct msqid_ds buf;

  /* Get the message queue for the id MY_MQ_ID */
  msgid = msgget( MY_MQ_ID, 0 );

  /* Check successful completion of msgget */
  if (msgid >= 0) {

    ret = msgctl( msgid, IPC_STAT, &buf );

    if (ret == 0) {

      printf( "Number of messages queued: %ld\n",
               buf.msg_qnum );
      printf( "Number of bytes on queue : %ld\n",
               buf.msg_cbytes );
      printf( "Limit of bytes on queue  : %ld\n",
               buf.msg_qbytes );

      printf( "Last message writer (pid): %d\n",
               buf.msg_lspid );
      printf( "Last message reader (pid): %d\n”,
               buf.msg_lrpid );

      printf( “Last change time         : %s”,
               ctime(&buf.msg_ctime) );

      if (buf.msg_stime) {
        printf( "Last msgsnd time         : %s”,
                 ctime(&buf.msg_stime) );
      }
      if (buf.msg_rtime) {
        printf( "Last msgrcv time         : %s”,
                 ctime(&buf.msg_rtime) );
      }

    }

  }

  return 0;
}
```

Listing 16.8 begins as most other message queue examples, with the collection of the message queue ID from `msgget`. After you have your ID, you use this to collect the message queue structure using `msgctl` and the command `IPC_STAT`. You pass in a reference to the `msqid_ds` structure, which is filled in by the `msgctl` API function. You then emit the information collected in lines 23–45.

At lines 23–24, you emit the number of messages that are currently enqueued on the message queue (`msg_qnum`). The current total number of bytes that are enqueued is identified by `msg_cbytes` (lines 25–26), and the maximum number of bytes that can be enqueued is defined by `msg_qbytes` (lines 27–28).

You can also identify the last reader and writer process IDs (lines 30–33). These refer to the effective process ID of the calling process that called `msgrcv` or `msgsnd`.

The `msg_ctime` element refers to the last time the message queue was changed (or when it was created). It's in standard `time_t` format, so you pass `msg_ctime` to `ctime` to grab the ASCII text version of the calendar date and time. You do the same for `msg_stime` (last `msgsnd` time) and `msg_rtime` (last `msgrcv` time). Note that in the case of `msg_stime` and `msg_rtime`, you emit the string dates only if their values are nonzero. If the values are zero, no `msgrcv` or `msgsnd` API functions have been called.

msgsnd

The `msgsnd` API function allows a process to send a message to a queue. As you saw in the introduction, the message is purely user-defined except that the first element in the message must be a `long` word for the type field. The function prototype for the `msgsnd` function is defined as follows:

```
int msgsnd( int msgid, struct msgbuf *msgp, size_t msgsz,
                  int msgflg );
```

The `msgid` argument is the message queue ID (returned from the `msgget` function). The `msgbuf` represents the message to be sent; at a minimum it is a `long` value representing the message type. The `msgsz` argument identifies the size of the `msgbuf` passed in to `msgsend`, in bytes. Finally, the `msgflag` argument allows you to alter the behavior of the `msgsnd` API function.

The `msgsnd` function has some default behavior that you should consider. If insufficient room exists on the message queue to write the message, the process is blocked until sufficient room exists. Otherwise, if room exists, the call succeeds immediately with a zero return to the caller.

Because you have already looked at some of the standard uses of `msgsnd`, here's your chance to look at some of the more specialized cases. The blocking behavior is desirable in most cases because it can be the most efficient. In some cases, you might want to try to send a message, and if you're unable (because of the insufficient space on the message queue), do something else. Take a look at this example in the following code snippet:

```
ret = msgsnd( msgid, (struct msgbuf *)&myMessage,
                   sizeof(myMessage), IPC_NOWAIT );
if (ret == 0) {
  // Message was successfully enqueued
} else {
  if (errno == EAGAIN) {
    // Insufficient space, do something else...
  }
}
```

The `IPC_NOWAIT` symbol (passed in as the `msgflags`) tells the `msgsnd` API function that if insufficient space exists, don't block but instead return immediately. You know this because an error was returned (indicated by the `-1` return value), and the `errno` variable was set to `EAGAIN`. Otherwise, with a zero return, the message was successfully enqueued on the message queue for the receiver.

While a message queue should not be deleted as long as processes pend on `msgsnd`, a special error return surfaces when this occurs. If a process is currently blocked on a `msgsnd` and the message queue is deleted, then a `-1` value is returned with an `errno` value set to `EIDRM`.

One final item to note on `msgsnd` involves the parameters that are modified when the `msgsnd` API call finishes. Table 16.3 lists the entire structure, but the items modified after successful completion of the `msgsnd` API function are listed in Table 16.5.

TABLE 16.5 Structure Updates after Successful `msgsnd` Completion

Parameter	Update
`msg_lspid`	Set to the process ID of the process that called `msgsnd`
`msg_qnum`	Incremented by one
`msg_stime`	Set to the current time

Note that the `msg_stime` *is the time that the message was enqueued and not the time that the* `msgsnd` *API function was called. This can be important if the* `msgsnd` *function blocks (because of a full message queue).*

msgrcv

Now you can focus on the last function in the message queue API. The `msgrcv` API function provides the means to read a message from the queue. The user provides a message buffer (filled in within `msgrcv`) and the message type of interest. The function prototype for `msgrcv` is defined as follows:

```
ssize_t msgrcv( int msgid, struct msgbuf *msgp, size_t msgsz,
                          long msgtyp, int msgflg );
```

The arguments passed to `msgrcv` include the `msgid` (message queue identifier received from `msgget`), a reference to a message buffer (`msgp`), the size of the buffer (`msgsz`), the message type of interest (`msgtyp`), and finally a set of flags (`msgflag`). The first three arguments are self-explanatory, so this section concentrates on the latter two: `msgtyp` and `msgflag`.

The `msgtyp` argument (message type) specifies to `msgrcv` the messages to be received. Each message within the queue contains a message type. The `msgtyp` argument to `msgrcv` defines that only those types of messages are sought. If no messages of that type are found, the calling process blocks until a message of the desired type is enqueued. Otherwise, the first message of the given type is returned to the caller. The caller could provide a zero as the `msgtyp`, which tells `msgrcv` to ignore the message type and return the first message on the queue. One exception to the message type request is discussed with `msgflg`.

The `msgflg` argument allows the caller to alter the default behavior of the `msgrcv` API function. As with `msgsnd`, you can instruct `msgrcv` not to block if no messages are waiting on the queue. This is done also with the `IPC_NOWAIT` flag. the previous paragraph discussed the use of `msgtyp` with a zero and nonzero value, but what if you were interested in any flag except a certain one? This can be accomplished by setting `msgtyp` with the undesired message type and setting the flag `MSG_EXCEPT` within `msgflg`. Finally, the use of flag `MSG_NOERROR` instructs `msgrcv` to ignore the size check of the incoming message and the available buffer passed from the user and simply truncate the message if the user buffer isn't large enough. All of the options for `msgtyp` are described in Table 16.6, and options for `msgflg` are shown in Table 16.7.

TABLE 16.6 `msgtyp` Arguments for `msgrcv`

`msgtyp`	Description
0	Read the first message available on the queue.
>0	If the `msgflg` `MSG_EXCEPT` is set, read the first message on the queue not equal to the `msgtyp`. Otherwise, if `MSG_EXCEPT` is not set, read the first message on the queue with the defined `msgtyp`.
<0	The first message on the queue that is less than or equal to the absolute value of `msgtyp` is returned.

TABLE 16.7 `msgflg` Arguments for `msgrcv`

Flag	Description
`IPC_NOWAIT`	Return immediately if no messages awaiting are of the given `msgtyp` (no blocking).
`MSG_EXCEPT`	Return first message available other than `msgtyp`.
`MSG_NOERROR`	Truncate the message if user buffer isn't of sufficient size.

When a message is read from the queue, the internal structure representing the queue is automatically updated as shown in Table 16.8.

TABLE 16.8 Structure Updates after Successful `msgsnd` Completion

Parameter	Update
`msg_lrpid`	Set to the process ID of process calling **`msgrcv`**
`msg_qnum`	Decremented by one
`msg_rtime`	Set to the current time

NOTE

Note that `msg_rtime` *is the time that the message was dequeued and not the time that the* `msgrcv` *API function was called. This can be important if the* `msgrcv` *function blocks (because of an empty message queue).*

Now take a look at some examples to illustrate msgrcv and the use of msgtyp and msgflg options. The most common use of msgrcv is to read the next available message from the queue:

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf), 0, 0 );
if (ret != -1) {
  printf("Message of type %ld received\n", *(long *)&buf );
}
```

Note that you check specifically for a return value that's not -1. You do this because msgrcv actually returns the number of bytes read. If the return is -1, errno contains the error that occurred.

If you desire not to block on the call, you can do this very simply as follows:

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf),
                  0, IPC_NOWAIT );
if (ret != -1) {
  printf("Message of type %ld received\n", *(long *)&buf );
} else if (errno == EAGAIN) {
  printf("Message unavailable\n");
}
```

With the presence of an error return from msgrcv and errno set to EAGAIN, it's understood that no messages are available for read. This isn't actually an error, just an indication that no messages are available in the nonblocking scenario.

Message queues permit multiple writers and readers to the same queue. These could be the same process, but very likely each is a different process. Say that you have a process that manages only a certain type of message. You identify this particular message by its message type. In the next example, you see a snippet from a process whose job it is to manage only messages of type 5.

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf), 5, 0 );
```

Any message sent of type 5 is received by the process executing this code snippet. To manage all other message types (other than 5), you can use the MSG_EXCEPT flag to receive these. Take for example:

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf),
                 5, MSG_EXCEPT );
```

Any message received on the queue other than type 5 is read using this line. If only messages of type 5 are available, this function blocks until a message not of type 5 is enqueued.

One final note on msgrcv is what happens if a process is blocked on a queue that is removed. The removal is permitted to occur, and the process blocked on the queue receives an error return with the errno set to EIDRM (as with blocked msgsnd calls). It's therefore important to fully recognize the error returns that are possible.

USER UTILITIES

GNU/Linux provides the ipcs command to explore IPC assets from the command line. The ipcs utility provides information on message queues as well as semaphores and shared memory segments. This section looks at its use for message queues.

The general form of the ipcs utility for message queues is as follows:

```
# ipcs -q
```

This presents all of the message queues that are visible to the process. You can start by creating a message queue (as was done in Listing 16.1):

```
# ./mqcreate
Created a Message Queue 819200
# ipcs -q

—— Message Queues ———
key        msqid      owner      perms      used-bytes   messages
0x0000006f 819200     mtj        666        0            0
```

You see the newly created queue (key 0x6f, or decimal 111). If you send a message to the message queue (such as was illustrated with Listing 16.4), you see the following:

```
# ./mqsend
Message successfully sent to queue 819200
# ipcs -q

—— Message Queues ———
key        msqid      owner      perms      used-bytes   messages
0x0000006f 819200     mtj        666        96           1
```

You see now that a message is contained on the queue that occupies 96 bytes. You can also take a deeper look at the queue by specifying the message queue ID. This is done with `ipcs` using the `-i` option:

```
# ipcs -q -i 819200
Message Queue msqid=819200
uid=500 gid=500 cuid=500       cgid=500       mode=0666
cbytes=96       qbytes=16384    qnum=1  lspid=22309     lrpid=0
send_time=Sat Mar 27 18:59:34 2004
rcv_time=Not set
change_time=Sat Mar 27 18:58:43 2004
```

You're now able to review the structure representing the message queue (as defined in Table 16.3). The `ipcs` utility can be very useful to view snapshots of message queues for application debugging.

You can also delete queues from the command line using the `ipcrm` command. To delete your previously created message queue, you simply use the `ipcrm` command as follows:

```
$ ipcrm -q 819200
$
```

As with the message queue API functions, you pass the message queue ID as the indicator of the message queue to remove.

SUMMARY

This chapter introduced the message queue API and its application of interprocess communication. It began with a whirlwind tour of the API and then detailed each of the functions, including the behavioral modifiers (`msgflg` arguments). Finally, it reviewed the `ipcs` utility and demonstrated its use as a debugging tool as well as the `ipcrm` command for removing message queues from the command line.

Message Queue APIs

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget( key_t key, int msgflg );
int msgctl( int msgid, int cmd, struct msqid_ds *buf );
int msgsnd( int msgid, structu msgbuf *msgp, size_t msgsz,
int msgflg );
size_t msgrcv( int msgid, struct msgbuf *msgp, size_t msgsz,
long msgtyp, int msgflg );
```

17 Synchronization with Semaphores

In This Chapter

- Introduction to GNU/Linux Semaphores
- Discussion of Binary and Counting Semaphores
- Creating and Configuring Semaphores
- Acquiring and Releasing Semaphores
- Single Semaphores or Semaphore Arrays
- The `ipcs` and `ipcrm` Utilities for Semaphores

INTRODUCTION

This chapter explores the topic of semaphores. GNU/Linux provides both binary and counting semaphores using the same POSIX-compliant API function set. It also investigates semaphores in GNU/Linux and their similarities with some of the other interprocess communication (IPC) mechanisms.

SEMAPHORE THEORY

First you need to go through a quick review of semaphore theory. A semaphore is nothing more than a variable that is protected. It provides a means to restrict access to a resource that is shared amongst two or more processes. Two operations are permitted, commonly called acquire and release. The acquire operation allows a process to take the semaphore, and if it has already been acquired, then the process blocks until it's available. If a process has the semaphore, it can release it, which allows other processes to acquire it. The process of releasing a semaphore automatically wakes up the next process awaiting it on the acquire operation. Consider the simple example in Figure 17.1.

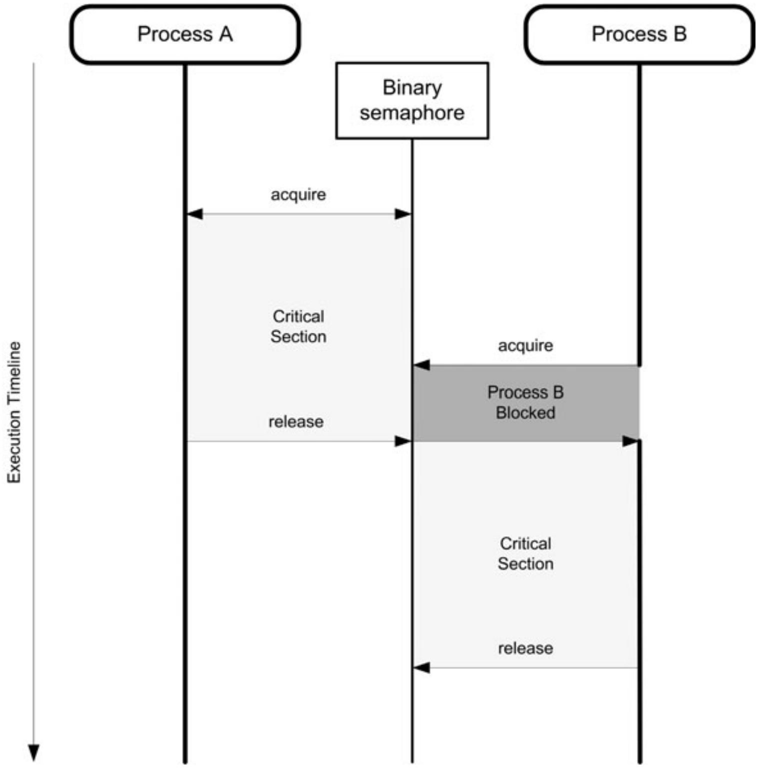

FIGURE 17.1 Simple binary semaphore example with two processes.

As shown in Figure 17.1, two processes are both vying for the single semaphore. Process A performs the acquire first and, therefore, is provided with the semaphore. The period in which the semaphore is owned by the process is commonly called a *critical section*. The critical section can be performed by only one process—hence the need for the coordination provided by the semaphore. While process A has the semaphore, process B is not permitted to perform its critical section.

Note that while process A is in its critical section, process B attempts to acquire the semaphore. As the semaphore has already been acquired by process A, process B is placed into a blocked state. When process A finally releases the semaphore, it is then granted to process B, which is allowed to enter its critical section. process B at a later time releases the semaphore, making it available for acquisition.

Semaphores commonly represent a point of synchronization in a system. For example, a semaphore can represent access to a shared resource. Only when the process has access to the semaphore can it access the shared resources. This ensures that only one process has access to the shared resource at a time, thus providing coordination between two or more users of the resource.

NOTE

Semaphores were invented by Edsger Dijkstra for the T.H.E. operating system. Originally, the semaphore operations were defined as P and V. The P stands for the Dutch proberen, or to test, and the V for verhogen, or to increment.

Edsger Dijkstra used the train analogy to illustrate the critical section. Imagine two parallel train tracks that for a short duration merge into a single track. The single track is the shared resource and is also the critical section. The semaphore ensures that only one train is permitted on the shared track at a time. Not having the semaphore can have disastrous results on two trains trying to use the shared track at the same time. The effect on software is just as treacherous.

TYPES OF SEMAPHORES

Semaphores come in two basic varieties. The first are *binary semaphores*, as illustrated in Figure 17.1. The binary semaphore represents a single resource; therefore, when one process has acquired it, others are blocked until it is released.

The other style is the *counting semaphore*, which is used to represent shared resources in quantities greater than one. Consider a pool of buffers. A counting semaphore can represent the entire set of buffers by setting its value to the number of buffers available. Each time a process requires a buffer, it acquires the semaphore, which decrements its value. When the semaphore value reaches zero, processes are blocked until the value becomes nonzero. When a semaphore is released, the semaphore value is increased, thus permitting other processes to acquire a semaphore (and associated buffer). This is the one use for a counting semaphore (see Figure 17.2).

In the counting semaphore example, each process requires two resources before being able to perform its desired activities. In this example, the value of the counting semaphore is 3, which means that only one process is permitted to fully operate at a time. Process A acquires its two resources first, which means that process B blocks until process A releases at least one of its resources.

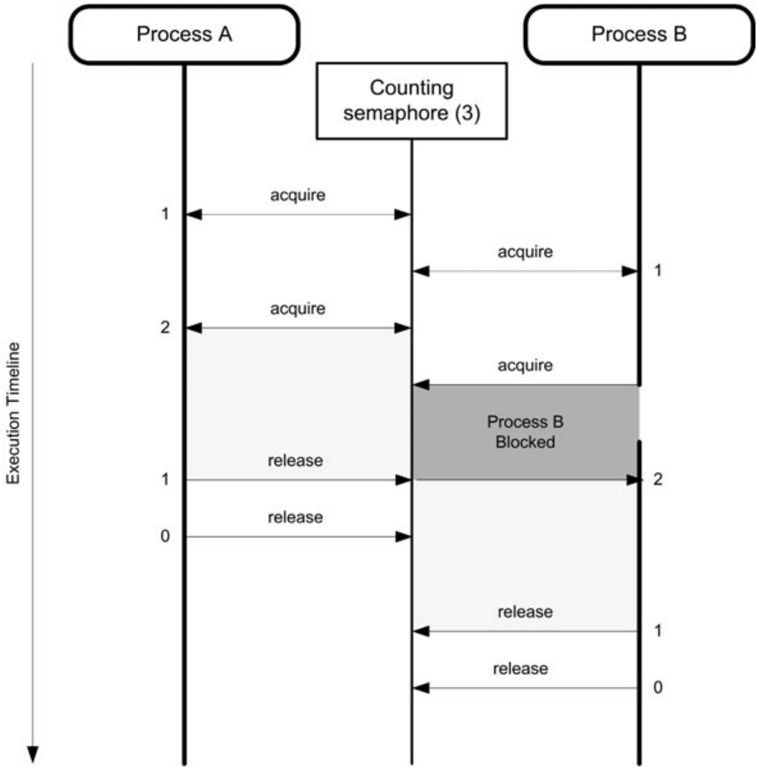

FIGURE 17.2 Counting semaphore example with two processes.

Quick Overview of GNU/Linux Semaphores

This chapter's discussion begins with a whirlwind tour of the GNU/Linux semaphore API. This section looks at code examples illustrating each of the API capabilities such as creating a new semaphore, finding a semaphore, acquiring a semaphore, releasing a semaphore, configuring a semaphore, and removing a semaphore. After you've finished the quick overview, you can dig deeper into the semaphore API.

NOTE

Semaphores in GNU/Linux are actually semaphore arrays. A single semaphore can represent an array of 64 semaphores. This unique feature of GNU/Linux permits atomic operations over numerous semaphores at the same time. In the early discussions of GNU/Linux semaphores in this chapter, you explore single semaphore uses. In the detailed discussions that follow, you look at the more complex semaphore array examples.

Using the semaphore API requires that the function prototypes and symbols be available to the application. This is done by including the following three header files:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Creating a Semaphore

To create a semaphore (or get an existing semaphore), you use the `semget` API function. This function takes a semaphore key, a semaphore count, and a set of flags. The count represents the number of semaphores in the set. In this case, you specify the need for one semaphore. The semaphore flags, argument 3 as shown in Listing 17.1, specify that the semaphore is to be created (`IPC_CREAT`). You also specify the read/write permissions to use (in this case `0666` for read/write for the user, group, and system in octal). An important item to consider is that when a semaphore is created, its value is zero. This suits for this example, but this chapter investigates later how to initialize the semaphore's value.

Listing 17.1 demonstrates creating a semaphore. In the following examples, you use the key `MY_SEM_ID` to represent your globally unique semaphore. At line 10, you use the `semget` with your semaphore key, semaphore set count, and command (with read/write permissions).

LISTING 17.1 Creating a Semaphore with `semget` (on the CD-ROM at `./source/ch17/semcreate.c`)

```
1:    #include <stdio.h>
2:    #include <sys/sem.h>
3:    #include "common.h"
4:
5:    int main()
6:    {
7:      int semid;
8:
9:      /* Create the semaphore with the id MY_SEM_ID */
10:     semid = semget( MY_SEM_ID, 1, 0666 | IPC_CREAT );
11:
12:     if (semid >= 0) {
13:
14:       printf( "semcreate: Created a semaphore %d\n", semid );
15:
16:     }
17:
```

```
    return 0;
  }
```

Upon completion of this simple application, a new globally available semaphore would be available with a key identified by `MY_SEM_ID`. Any process in the system could use this semaphore.

GETTING AND RELEASING A SEMAPHORE

Now take a look at an application that attempts to acquire an existing semaphore and also another application that releases it. Recall that your previously created semaphore (in Listing 17.1) was initialized with a value of zero. This is identical to a binary semaphore already having been acquired.

Listing 17.2 illustrates an application acquiring your semaphore. The GNU/Linux semaphore API is a little more complicated than many semaphore APIs, but it is POSIX compliant and, therefore, important for porting to other UNIX-like operating systems.

LISTING 17.2 Getting a Semaphore with `semop`

```
#include <stdio.h>
#include <sys/sem.h>
#include <stdlib.h>
#include "common.h"

int main()
{
  int semid;
  struct sembuf sb;

  /* Get the semaphore with the id MY_SEM_ID */
  semid = semget( MY_SEM_ID, 1, 0 );

  if (semid >= 0) {

    sb.sem_num = 0;
    sb.sem_op = -1;
    sb.sem_flg = 0;

    printf( "semacq: Attempting to acquire semaphore %d\n", semid );

```

```
22:         /* Acquire the semaphore */
23:         if ( semop( semid, &sb, 1 ) == -1 ) {
24:
25:           printf( "semacq: semop failed.\n" );
26:           exit(-1);
27:
28:         }
29:
30:         printf( "semacq: Semaphore acquired %d\n", semid );
31:
32:       }
33:
34:       return 0;
35:     }
```

You begin by identifying the semaphore identifier with `semget` at line 12. If this is successful, you build your semaphore operations structure (identified by the `sembuf` structure). This structure contains the semaphore number, the operation to be applied to the semaphore, and a set of operation flags. Because you have only one semaphore, you use the semaphore number zero to identify it. To acquire the semaphore, you specify an operation of `-1`. This subtracts one from the semaphore, but only if it's greater than zero to begin with. If the semaphore is already zero, the operation (and the process) blocks until the semaphore value is incremented.

With the `sembuf` created (variable `sb`), you use this with the API function `semop` to acquire the semaphore. You specify the semaphore identifier, your `sembuf` structure, and then the number of `sembuf`s that were passed in (in this case, one). This implies that you can provide an array of `sembufs`, which is investigated later in the chapter. As long as the semaphore operation can finish (semaphore value is nonzero), then it returns with success (a non –1 value). This means that the process performing the `semop` has acquired the semaphore.

Now it's time to look at a release example. This example demonstrates the `semop` API function from the perspective of releasing the semaphore (see Listing 17.3).

NOTE

In many cases, the release follows the acquire in the same process. This usage allows synchronization between two processes. The first process attempts to acquire the semaphore and then blocks when it's not available. The second process, knowing that another process is sitting blocked on the semaphore, releases it, allowing the process to continue. This provides a lock-step operation between the processes and is practical and useful.

LISTING 17.3 Releasing a Semaphore with `semop` (on the CD-ROM at `./source/ch17/semrel.c`)

```
1:   #include <stdio.h>
2:   #include <sys/sem.h>
3:   #include <stdlib.h>
4:   #include "common.h"
5:
6:   int main()
7:   {
8:     int semid;
9:     struct sembuf sb;
10:
11:    /* Get the semaphore with the id MY_SEM_ID */
12:    semid = semget( MY_SEM_ID, 1, 0 );
13:
14:    if (semid >= 0) {
15:
16:      printf( "semrel: Releasing semaphore %d\n", semid );
17:
18:      sb.sem_num = 0;
19:      sb.sem_op  = 1;
20:      sb.sem_flg = 0;
21:
22:      /* Release the semaphore */
23:      if (semop( semid, &sb, 1 ) == -1) {
24:
25:        printf("semrel: semop failed.\n");
26:        exit(-1);
27:
28:      }
29:
30:      printf( "semrel: Semaphore released %d\n", semid );
31:
32:    }
33:
34:    return 0;
35:  }
```

At line 12 of Listing 17.3, you first identify the semaphore of interest using the `semget` API function. Having your semaphore identifier, you build your `sembuf` structure to release the semaphore at line 23 using the `semop` API function. In this example, your `sem_op` element is 1 (compared to the –1 in Listing 17.2). In this example, you are releasing the semaphore, which means that you're making it nonzero (and thus available).

> **NOTE**
>
> *It's important to note the symmetry the* `sembuf` *uses in Listings 17.2 and 17.3. To acquire the semaphore, you subtract 1 from its value. To release the semaphore, you add 1 to its value. When the semaphore's value is zero, it's unavailable, forcing any processing attempting to acquire it to block. An initial value of 1 for the semaphore defines it as a binary semaphore. If the semaphore value is greater than zero, it can be considered a counting semaphore.*

Now take a look at a sample application of each of the functions discussed thus far. Listing 17.4 illustrates execution of Listing 17.1, `semcreate`, Listing 17.2, `semacq`, and Listing 17.3, `semrel`.

LISTING 17.4 Execution of the Sample Semaphore Applications

```
1:      # ./semcreate
2:      semcreate: Created a semaphore 1376259
3:      # ./semacq &
4:      [1] 12189
5:      semacq: Attempting to acquire semaphore 1376259
6:      # ./semrel
7:      semrel: Releasing semaphore 1376259
8:      semrel: Semaphore released 1376259
9:      # semacq: Semaphore acquired 1376259
10:
11:     [1]+  Done                    ./semacq
12:     #
```

At line 1, you create the semaphore. You emit the identifier associated with this semaphore, `1376259` (which is shown at line 2). Next, at line 3, you perform the `semacq` application, which acquires the semaphore. You run this in the background (identified by the trailing `&` symbol) because this application immediately blocks because the semaphore is unavailable. At line 4, you see the creation of the new subprocess (where `[1]` represents the number of subprocesses and `12189` is its process ID, or pid). The `semacq` application prints out its message, indicating that it's attempting to acquire the semaphore, but then it blocks. You then execute the `semrel` application to release the semaphore (line 6). You see two messages from this application; the first at line 7 indicates that it is about to release the semaphore, and then at line 8, you see that it was successful. Immediately thereafter, you see the `semacq` application acquires the newly released semaphore, given its output at line 9.

Finally, at line 11, you see the `semacq` application subprocess finish. Because it is unblocked (based upon the presence of its desired semaphore), the `semacq`'s main function reached its return, and thus the process finished.

CONFIGURING A SEMAPHORE

While a number of elements can be configured for a semaphore, this section looks specifically at reading and writing the value of the semaphore (the current count).

The first example, Listing 17.5, demonstrates reading the current value of the semaphore. You achieve this using the `semctl` API function.

LISTING 17.5 Retrieving the Current Semaphore Count (on the CD-ROM at `./source/ch17/semcrd.c`)

```
#include <stdio.h>
#include <sys/sem.h>
#include <stdlib.h>
#include "common.h"

int main()
{
  int semid, cnt;

  /* Get the semaphore with the id MY_SEM_ID */
  semid = semget( MY_SEM_ID, 1, 0 );

  if (semid >= 0) {

    /* Read the current semaphore count */
    cnt = semctl( semid, 0, GETVAL );

    if (cnt != -1) {

      printf("semcrd: current semaphore count %d.\n", cnt);

    }

  }

  return 0;
}
```

Reading the semaphore count is performed at line 16. You specify the semaphore identifier, the index of the semaphore (`0`), and the command (`GETVAL`). Note that the semaphore is identified by an index because it can possibly represent an array of semaphores (rather than one). The return value from this command is either –1 for error or the count of the semaphore.

You can configure a semaphore with a count using a similar mechanism (as shown in Listing 17.6).

LISTING 17.6 Setting the Current Semaphore Count

```
1:    #include <stdio.h>
2:    #include <sys/sem.h>
3:    #include <stdlib.h>
4:    #include "common.h"
5:
6:    int main()
7:    {
8:      int semid, ret;
9:
10:     /* Get the semaphore with the id MY_SEM_ID */
11:     semid = semget( MY_SEM_ID, 1, 0 );
12:
13:     if (semid >= 0) {
14:
15:       /* Read the current semaphore count */
16:       ret = semctl( semid, 0, SETVAL, 6 );
17:
18:       if (ret != -1) {
19:
20:         printf( "semcrd: semaphore count updated.\n" );
21:
22:       }
23:
24:     }
25:
26:     return 0;
27:   }
```

As with retrieving the current semaphore value, you can set this value using the `semctl` API function. The difference here is that along with the semaphore identifier (`semid`) and semaphore index (`0`), you specify the set command (`SETVAL`) and a value. In this example (line 16 of Listing 17.6), you are setting the semaphore value to 6. Setting the value to 6, as shown here, changes the binary semaphore to a counting semaphore. This means that six semaphore acquires are permitted before an acquiring process blocks.

REMOVING A SEMAPHORE

Removing a semaphore is also performed through the `semctl` API function. After retrieving the semaphore identifier (line 10 in Listing 17.7), you remove the semaphore using the `semctl` API function and the `IPC_RMID` command (at line 14).

LISTING 17.7 Removing a Semaphore

```
1:   #include <stdio.h>
2:   #include <sys/sem.h>
3:   #include "common.h"
4:
5:   int main()
6:   {
7:     int semid, ret;
8:
9:     /* Get the semaphore with the id MY_SEM_ID */
10:    semid = semget( MY_SEM_ID, 1, 0 );
11:
12:    if (semid >= 0) {
13:
14:      ret = semctl( semid, 0, IPC_RMID);
15:
16:      if (ret != -1) {
17:
18:        printf( "Semaphore %d removed.\n", semid );
19:
20:      }
21:
22:    }
23:
24:    return 0;
25:  }
```

As you can probably see, the semaphore API probably is not the simplest that you've used before.

That's it for the whirlwind tour; next the chapter explores the semaphore API in greater detail and looks at some of its other capabilities.

THE SEMAPHORE API

As noted before, the semaphore API handles not only the case of managing a single semaphore, but also groups (or arrays) of semaphores. This section investigates the use of those groups of semaphores. As a quick review, Table 17.1 shows the API functions and describes their uses. The following discussion continues to use the term *semaphore*, but note this can refer instead to a semaphore array.

TABLE 17.1 Semaphore API Functions and Their Uses

API Function	Uses
`semget`	Create a new semaphore.
	Get an existing semaphore.
`semop`	Acquire or release a semaphore.
`semctl`	Get info about a semaphore.
	Set info about a semaphore.
	Remove a semaphore.

The following sections address each of these functions using both simple examples (a single semaphore) and the more complex uses (semaphore arrays).

semget

The `semget` API function serves two fundamental roles. Its first use is in the creation of new semaphores. The second use is identifying an existing semaphore. In both cases, the response from `semget` is a semaphore identifier (a simple integer value representing the semaphore). The prototype for the `semget` API function is defined as follows:

```
int semget( key_t key, int nsems, int semflg );
```

The `key` argument specifies a system-wide identifier that uniquely identifies this semaphore. The key must be nonzero or the special symbol `IPC_PRIVATE`. The `IPC_PRIVATE` variable tells `semget` that no key is provided and to simply make one up. Because no key exists, other processes have no way to know about this semaphore. Therefore, it's a private semaphore for this particular process.

You can create a single semaphore (with an `nsems` value of 1) or multiple semaphores. If you're using `semget` to get an existing semaphore, this value can simply be zero.

Finally, the `semflg` argument allows you to alter the behavior of the `semget` API function. The `semflg` argument can take on three basic forms, depending upon what you desire. In the first form, you want to create a new semaphore. In this case, the `semflg` argument must be the `IPC_CREAT` value OR'd with the permissions (see Table 17.2). The second form also provides for semaphore creation, but with the constraint that if the semaphore already exists, an error is generated. This second form requires the `semflg` argument to be set to `IPC_CREAT | IPC_EXCL` along with the permissions. If the second form is used and the semaphore already exists, the call fails (–1 return) with `errno` set to `EEXIST`. The third form takes a zero for `semflg` and identifies that an existing semaphore is being requested.

TABLE 17.2 Semaphore Permissions for the `semget semflg` Argument

Symbol	Value	Meaning
S_IRUSR	0400	User has read permission.
S_IWUSR	0200	User has write permission.
S_IRGRP	0040	Group has read permission.
S_IWGRP	0020	Group has write permission.
S_IROTH	0004	Other has read permission.
S_IWOTH	0002	Other has write permission.

Now it's time to look at a few examples of `semget`, used in each of the three scenarios defined earlier in this section. In the examples that follow, assume `semid` is an `int` value, and `mySem` is of type `key_t`. In the first example, you create a new semaphore (or access an existing one) of the private type.

```
semid = semget( IPC_PRIVATE, 1, IPC_CREAT | 0666 );
```

After the `semget` call completes, the semaphore identifier is stored in `semid`. Otherwise, if an error occurs, a –1 is returned. Note that in this example (using `IPC_PRIVATE`), `semid` is all you have to identify this semaphore. If `semid` is somehow lost, you have no way to find this semaphore again.

In the next example, you create a semaphore using a system-wide unique key value (0x222). You also indicate that if the semaphore already exists, you don't simply get its value, but instead fail the call. Recall that this is provided by the IPC_EXCL command, as follows:

```
// Create a new semaphore
semid = semget( 0x222, 1, IPC_CREAT | IPC_EXCL | 0666 );
if ( semid == -1 ) {
  printf( "Semaphore already exists, or error\n" );
} else {
  printf( "Semaphore created (id %d)\n", semid );
}
```

If you don't want to rely on the fact that 0x222 might not be unique in your system, you can use the ftok system function. This function provides the means to create a new unique key in the system. It does this by using a known file in the filesystem and an integer number. The file in the filesystem is unique by default (considering its path). Therefore, by using the unique file (and integer), it's an easy task to then create a unique system-wide value. Take a look at an example of the use of ftok to create a unique key value. Assume for this example that your file and path are defined as /home/mtj/semaphores/mysem.

```
key_t mySem;
int   semid;
// Create a key based upon the defined path and number
myKey = ftok( "/home/mtj/semaphores/mysem", 0 );
semid = semget( myKey, 1, IPC_CREAT | IPC_EXCL | 0666 );
```

Note that each time ftok is called with those parameters, the same key is generated (which is why this method works at all!). As long as each process that needs access to the semaphore knows about the file and number, the key can be recalculated and then used to identify the semaphore.

In the examples discussed thus far, you've created a single semaphore. You can create an array of semaphores by simply specifying an nsems value greater than one, such as the following:

```
semarrayid = semget( myKey, 10, IPC_CREAT | 0666 );
```

The result is a semaphore array created that consists of 10 semaphores. The return value (semarrayid) represents the entire set of semaphores. You get a chance to see how individual semaphores can be addressed in the semctl and semop discussions later in the chapter.

In this last example of semget, you simply get the semaphore identifier of an existing semaphore. In this example, you specify the key value and no command:

```
semid = semget( 0x222, 0, 0 );
if ( semid == -1 ) {
  printf( "Semaphore does not exist...\n" );
}
```

One final note on semaphores is that, just as is the case with message queues, a set of defaults is provided to the semaphore as it's created. The parameters that are defined are shown in Table 17.3. Later on in the discussion of semctl, you can see how some of the parameters can be changed.

TABLE 17.3 Semaphore Internal Values

Parameter	Default Value
sem_perm.cuid	Effective user ID of the calling process (creator)
sem_perm.uid	Effective user ID of the calling process (owner)
sem_perm.cgid	Effective group ID of the calling process (creator)
sem_perm.gid	Effective group ID of the calling process (owner)
sem_perm.mode	Permissions (lower 9 bits of semflg)
sem_nsems	Set to the value of nsems
sem_otime	Set to zero (last semop time)
sem_ctime	Set to the current time (create time)

The process can override some of these parameters. You get to explore this later in the discussion of semctl.

semctl

The semctl API function provides a number of control operations on semaphores or semaphore arrays. Examples of functionality range from setting the value of the semaphore (as shown in Listing 17.6) to removing a semaphore or semaphore array (see Listing 17.7). You get a chance to see these and other examples in this section.

The function prototype for the semctl call is as follows:

```
int semctl( int semid, int semnum, int cmd, ... );
```

The first argument defines the semaphore identifier, the second defines the semaphore number of interest, the third defines the command to be applied, and then potentially another argument (usually defined as a union). The operations that can be performed are shown in Table 17.4.

TABLE 17.4 Operations That Can Be Performed Using `semctl`

Command	Description	Fourth Argument
`GETVAL`	Return the semaphore value.	
`SETVAL`	Set the semaphore value.	`int`
`GETPID`	Return the process ID that last operated on the semaphore (`semop`).	
`GETNCNT`	Return the number of processes awaiting the defined semaphore to increase in value.	`int`
`GETZCNT`	Return the number of processes awaiting the defined semaphore to become zero.	`int`
`GETALL`	Return the value of each semaphore in a semaphore array.	`u_short*`
`SETALL`	Set the value of each semaphore in a semaphore array.	`u_short*`
`IPC_STAT`	Return the effective user, group, and permissions for a semaphore.	`struct semid_ds*`
`IPC_SET`	Set the effective user, group, and permissions for a semaphore.	`struct semid_ds*`
`IPC_RMID`	Remove the semaphore or semaphore array.	

Now it's time to look at some examples of each of these operations in `semctl`, focusing on semaphore array examples where applicable. The first example illustrates the setting of a semaphore value and then returning its value. In this example, you first set the value of the semaphore to `10` (using the command `SETVAL`) and then read it back out using `GETVAL`. Note that the `semnum` argument (argument 2) defines an individual semaphore. Later on, you can take look at the semaphore array case with `GETALL` and `SETALL`.

```
int semid, ret, value;
...
/* Set the semaphore to 10 */
ret = semctl( semid, 0, SETVAL, 10 );
...
/* Read the semaphore value (return value) */
value = semctl( semid, 0, GETVAL );
```

The GETPID command allows you to identify the last process that performed a semop on the semaphore. The process identifier is the return value, and argument 4 is not used in this case.

```
int semid, pid;
...
pid = semctl( semid, 0, GETPID );
```

If no semop has been performed on the semaphore, the return value is zero.

To identify the number of semaphores that are currently awaiting a semaphore to increase in value, you can use the GETNCNT command. You can also identify the number of processes that are awaiting the semaphore value to become zero using GETZCNT. Both of these commands are illustrated in the following for the semaphore numbered zero:

```
int semid, count;
/* How many processes are awaiting this semaphore to increase */
count = semctl( semid, 0, GETNCNT );
/* How many processes are awaiting this semaphore to become zero */
count = semctl( semid, 0, GETZCNT );
```

Now it's time to take a look at an example of some semaphore array operations. Listing 17.8 illustrates both the SETVAL and GETVAL commands with semctl.

In this example, you create a semaphore array of 10 semaphores. The creation of the semaphore array is performed at lines 20–21 using the semget API function. Note that because you're going to create and remove the semaphore array within this same function, you use no key and instead use the IPC_PRIVATE key. The MAX_SEMAPHORES symbol defines the number of semaphores that you are going to create, and finally you specify that you are creating the semaphore array (IPC_CREAT) with the standard permissions.

Next, you initialize the semaphore value array (lines 26–30). While this is not a traditional example, you initialize each semaphore to one plus its semnum (so semaphore zero has a value of one, semaphore one has a value of two, and so on). You do this so that you can inspect the value array later and know what you're looking at. At line 33, you set the `arg.array` parameter to the address of the array (`sem_array`). Note that you're using the `semun` union, which defines some commonly used types for semaphores. In this case, you use the `unsigned short` field to represent an array of semaphore values.

At line 36, you use the `semctl` API function and the `SETALL` command to set the semaphore values. You provide the semaphore identifier `semnum` as zero (unused in this case), the `SETALL` command, and finally the `semun` union. Upon return of this API function, the semaphore array identified by `semid` has the values as defined in `sem_array`.

Next, you explore the `GETALL` command, which retrieves the array of values for the semaphore array. You first set your `arg.array` to a new array (just to avoid reusing the existing array that has the contents that you are looking for), at line 41. At line 44, you call `semctl` again with the `semid`, zero for `semnum` (unused here, again), the `GETALL` command, and the `semun` union.

To illustrate what you have read, you next loop through the `sem_read_array` and emit each value for each semaphore index within the semaphore array (lines 49–53).

While `GETALL` allows you to retrieve the entire semaphore array in one call, you can perform the same action using the `GETVAL` command, calling `semctl` for each semaphore of the array. This is illustrated at lines 56–62. This also applies to using the `SETVAL` command to mimic the `SETALL` behavior.

Finally, at line 65, you use the `semctl` API function with the `IPC_RMID` command to remove the semaphore array.

LISTING 17.8 Creating and Manipulating Semaphore Arrays (on the CD-ROM at `./source/ch17/semall.c`)

```
1:      #include <stdio.h>
2:      #include <sys/types.h>
3:      #include <sys/sem.h>
4:      #include <errno.h>
5:
6:      #define MAX_SEMAPHORES  10
7:
8:      int main()
9:      {
10:       int i, ret, semid;
11:       unsigned short sem_array[MAX_SEMAPHORES];
12:       unsigned short sem_read_array[MAX_SEMAPHORES];
13:
14:       union semun {
15:         int val;
```

```
    struct semid_ds *buf;
    unsigned short *array;
  } arg;

  semid = semget( IPC_PRIVATE, MAX_SEMAPHORES,
                  IPC_CREAT | 0666 );

  if (semid != -1) {

    /* Initialize the sem_array */
    for ( i = 0 ; i < MAX_SEMAPHORES ; i++ ) {

      sem_array[i] = (unsigned short)(i+1);

    }

    /* Update the arg union with the sem_array address */
    arg.array = sem_array;

    /* Set the values of the semaphore-array */
    ret = semctl( semid, 0, SETALL, arg );

    if (ret == -1) printf("SETALL failed (%d)\n", errno);

    /* Update the arg union with another array for read */
    arg.array = sem_read_array;

    /* Read the values of the semaphore array */
    ret = semctl( semid, 0, GETALL, arg );

    if (ret == -1) printf("GETALL failed (%d)\n", errno);

    /* print the sem_read_array */
    for ( i = 0 ; i < MAX_SEMAPHORES ; i++ ) {

      printf("Semaphore %d, value %d\n", i,
      sem_read_array[i] );

    }

    /* Use GETVAL in a similar manner */
    for ( i = 0 ; i < MAX_SEMAPHORES ; i++ ) {

      ret = semctl( semid, i, GETVAL );

```

```
            printf("Semaphore %d, value %d\n", i, ret );

          }

          /* Delete the semaphore */
          ret = semctl( semid, 0, IPC_RMID );

        } else {

          printf("Could not allocate semaphore (%d)\n", errno);

        }

        return 0;
      }
```

Executing this application (called `semall`) produces the output shown in Listing 17.9. Not surprisingly, the `GETVAL` emits identical output as that shown for the `GETALL`.

LISTING 17.9 Output from the `semall` Application Shown in Listing 17.8

```
# ./semall
Semaphore 0, value 1
Semaphore 1, value 2
Semaphore 2, value 3
Semaphore 3, value 4
Semaphore 4, value 5
Semaphore 5, value 6
Semaphore 6, value 7
Semaphore 7, value 8
Semaphore 8, value 9
Semaphore 9, value 10
Semaphore 0, value 1
Semaphore 1, value 2
Semaphore 2, value 3
Semaphore 3, value 4
Semaphore 4, value 5
Semaphore 5, value 6
Semaphore 6, value 7
Semaphore 7, value 8
Semaphore 8, value 9
Semaphore 9, value 10
#
```

The `IPC_STAT` command retrieves the current information about a semaphore or semaphore array. The data is retrieved into a structure called `semid_ds` and contains a variety of parameters. The application that reads this information is shown in Listing 17.10. You read the semaphore information at line 23 using the `semctl` API function and the `IPC_STAT` command. The information captured is then emitted at lines 27–49.

LISTING 17.10 Reading Semaphore Information Using `IPC_STAT` (on the CD-ROM at `./source/ch17/semstat.c`)

```
1:     #include <stdio.h>
2:     #include <sys/sem.h>
3:     #include <time.h>
4:     #include "common.h"
5:
6:     int main()
7:     {
8:       int semid, ret;
9:       struct semid_ds sembuf;
10:
11:      union semun {
12:        int val;
13:        struct semid_ds *buf;
14:        unsigned short *array;
15:      } arg;
16:
17:      /* Get the semaphore with the id MY_SEM_ID */
18:      semid = semget( MY_SEM_ID, 1, 0 );
19:
20:      if (semid >= 0) {
21:
22:        arg.buf = &sembuf;
23:        ret = semctl( semid, 0, IPC_STAT, arg );
24:
25:        if (ret != -1) {
26:
27:          if (sembuf.sem_otime) {
28:            printf( "Last semop time %s",
29:                     ctime( &sembuf.sem_otime ) );
30:          }
31:
```

```
            printf( "Last change time %s",
                     ctime( &sembuf.sem_ctime ) );

            printf( "Number of semaphores %ld\n”,
                     sembuf.sem_nsems );

            printf( "Owner’s user id %d\n",
                     sembuf.sem_perm.uid );
            printf( "Owner’s group id %d\n",
                     sembuf.sem_perm.gid );

            printf( "Creator’s user id %d\n",
                     sembuf.sem_perm.cuid );
            printf( "Creator’s group id %d\n",
                     sembuf.sem_perm.cgid );

            printf( "Permissions 0%o\n",
                     sembuf.sem_perm.mode );

          }

        }

        return 0;
      }
```

Three of the fields shown can be updated through another call to `semctl` using the `IPC_SET` call. The three updateable parameters are the effective user ID (`sem_perm.uid`), the effective group ID (`sem_perm.gid`), and the permissions (`sem_perm.mode`). The following code snippet illustrates modifying the permissions:

```
/* First, read the semaphore information */
arg.buf = &sembuf;
ret = semctl( semid, 0, IPC_STAT, arg );
/* Next, update the permissions */
sembuf.sem_perm.mode = 0644;
/* Finally, update the semaphore information */
ret = semctl( semid, 0, IPC_SET, arg );
```

After the `IPC_SET semctl` has completed, the last change time (`sem_ctime`) is updated to the current time.

Finally, the IPC_RMID command permits you to remove a semaphore or semaphore array. A code snippet demonstrating this process is shown in the following:

```
int semid;
...
semid = semget( the_key, NUM_SEMAPHORES, 0 );
ret = semctl( semid, 0, IPC_RMID );
```

Note that if any processes are currently blocked on the semaphore, they are immediately unblocked with an error return and errno is set to EIDRM.

semop

The semop API function provides the means to acquire and release a semaphore or semaphore array. The basic operations provided by semop are to decrement a semaphore (acquire one or more semaphores) or to increment a semaphore (release one or more semaphores). The API for the semop function is defined as follows:

```
int semop( int semid, struct sembuf *sops, unsigned int nsops );
```

The semop takes three parameters: a semaphore identifier (semid), a sembuf structure, and the number of semaphore operations to be performed (nsops). The semaphore structure defines the semaphore number of interest, the operation to perform, and a flag word that can be used to alter the behavior of the operation. The sembuf structure is shown as follows:

```
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
};
```

As you can imagine, the sembuf array can produce very complex semaphore interactions. You can acquire one semaphore and release another in a single semop operation.

Take a look at a simple application that acquires 10 semaphores in one operation. This application is shown in Listing 17.11.

An important difference to notice here is that rather than specify a single sembuf structure (as you did in single semaphore operations), you specify an array of sembufs (line 9). You identify your semaphore array at line 12; note again that you specify the number of semaphores (nsems, or number of semaphores, as argument 2). You build out your sembuf array as acquires (with a sem_op of –1) and also initialize the `sem_num` field with the semaphore number. This specifies that you want to acquire each of the semaphores in the array. If one or more aren't available, the operation blocks until all semaphores can be acquired.

At line 26, you perform the `semop` API function to acquire the semaphores. Upon acquisition (or error), the `semop` function returns to the application. As long as the return value is not –1, you have successfully acquired the semaphore array. Note that you can specify –2 for each `sem_op`, which requires that two counts of the semaphore are needed for successful acquisition.

LISTING 17.11 Acquiring an Array of Semaphores Using `semop` (on the CD-ROM at `./source/ch17/semaacq.c`)

```
1:    #include <stdio.h>
2:    #include <sys/sem.h>
3:    #include <stdlib.h>
4:    #include "common.h"
5:
6:    int main()
7:    {
8:      int semid, i;
9:      struct sembuf sb[10];
10:
11:     /* Get the semaphore with the id MY_SEM_ID */
12:     semid = semget( MY_SEMARRAY_ID, 10, 0 );
13:
14:     if (semid >= 0) {
15:
16:       for (i = 0 ; i < 10 ; i++) {
17:         sb[i].sem_num = i;
18:         sb[i].sem_op = -1;
19:         sb[i].sem_flg = 0;
20:       }
21:
22:       printf( "semaacq: Attempting to acquire semaphore %d\n",
23:                semid );
24:
25:       /* Acquire the semaphores */
26:       if (semop( semid, &sb[0], 10 ) == -1) {
27:
28:         printf("semaacq: semop failed.\n");
29:         exit(-1);
30:
31:       }
```

```

        printf( "semaacq: Semaphore acquired %d\n", semid );

      }

      return 0;
}
```

Next, take a look at the semaphore release operation. This includes only the changes from Listing 17.11, as otherwise they are very similar (on the CD-ROM at `./source/ch17/semarel.c`). In fact, the only difference is the `sembuf` initialization:

```
for ( i = 0 ; i < 10 ; i++ ) {
  sb[i].sem_num = i;
  sb[i].sem_op = 1;
  sb[i].sem_flg = 0;
}
```

In this example, you increment the semaphore (release) instead of decrementing it (as was done in Listing 17.11).

The `sem_flg` within the `sembuf` structure permits you to alter the behavior of the `semop` API function. Two flags are possible, as shown in Table 17.5.

TABLE 17.5 Semaphore Flag Options (`sembuf.sem_flg`)

Flag	Purpose
`SEM_UNDO`	Undo the semaphore operation if the process exits.
`IPC_NOWAIT`	Return immediately if the semaphore operation cannot be performed (if the process would block) and return an `errno` of `EAGAIN`.

Another useful operation that can be performed on semaphores is the wait-for-zero operation. In this case, the process is blocked until the semaphore value becomes zero. This operation is performed by simply setting the `sem_op` field to zero, as follows:

```
struct sembuf sb;
...
sb.sem_num = 0;        // semaphore 0
sb.sem_op = 0;        // wait for zero
sb.sem_flg = 0;        // no flags
...
```

As with previous `semops`, setting `sem_flg` with `IPC_NOWAIT` causes `semop` to return immediately if the operation blocks with an `errno` of `EAGAIN`.

Finally, if the semaphore is removed while a process is blocked on it (via a `semop` operation), the process becomes immediately unblocked and an `errno` value is returned as `EIDRM`.

User Utilities

GNU/Linux provides the `ipcs` command to explore semaphores from the command line. The `ipcs` utility provides information on a variety of resources; this section explores its use for investigating semaphores.

The general form of the `ipcs` utility for semaphores is as follows:

```
# ipcs -s
```

This presents all the semaphores that are visible to the calling process. Take a look at an example where you create a semaphore (as was done in Listing 17.1):

```
# ./semcreate
semcreate: Created a semaphore 1769475
# ipcs -s

—— Semaphore Arrays ———
key        semid      owner      perms      nsems
0x0000006f 1769475    mtj       666        1

#
```

Here, you see your newly created semaphore (`key 0x6f`). You can get extended information about the semaphore using the `-i` option. This allows you to specify a specific semaphore ID, for example:

```
# ipcs -s -i 1769475

Semaphore Array semid=1769475
uid=500  gid=500        cuid=500        cgid=500
mode=0666, access_perms=0666
nsems = 1
otime = Not set
ctime = Fri Apr  9 17:50:01 2004
semnum     value      ncount     zcount     pid
0          0          0          0          0

#
```

Here you see your semaphore in greater detail. You see the owner and creator process and group IDs, permissions, number of semaphores (`nsems`), last `semop` time, last change time, and the details of the semaphore itself (`semnum` through `pid`). The value represents the actual value of the semaphore (zero after creation). If you were to perform the release operation (see Listing 17.3) and then perform this command again, you would then see this:

```
# ./semrel
    semrel: Releasing semaphore 1769475
    semrel: Semaphore released 1769475
    # ipcs -s -i 1769475

    Semaphore Array semid=1769475
    uid=500  gid=500        cuid=500        cgid=500
    mode=0666, access_perms=0666
    nsems = 1
    otime = Fri Apr  9 17:54:44 2004
    ctime = Fri Apr  9 17:50:01 2004
    semnum     value      ncount     zcount     pid
    0          1          0          0          20494

    #
```

Note here that your value has increased (based upon the semaphore release), and other information (such as `otime` and `pid`) has been updated given a semaphore operation having been performed.

You can also delete semaphores from the command line using the `ipcrm` command. To delete your previously created semaphore, you simply use the `ipcrm` command as follows:

```
# ipcrm -s 1769475
[mtj@camus ch17]$ ipcs -s

—— Semaphore Arrays ———
key        semid      owner      perms      nsems

#
```

As with the semop and semctl API functions, the ipcrm command uses the semaphore identifier to specify the semaphore to be removed.

Summary

This chapter introduced the semaphore API and its application of interprocess coordination and synchronization. It began with a whirlwind tour of the API and then followed with a detailed description of each command including examples of each. Finally, the chapter reviewed the ipcs and ipcrm commands and demonstrated their debugging and semaphore management capabilities.

Semaphore APIs

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget( key_t key, int nsems, int semflg );
int semop( int semid, struct sembuf *sops, unsigned int nsops );
int semctl( int semid, int semnum, int cmd, ... );
```