5

Processes and Threads

5.1 Introduction

We now leave the subject of input and output to begin investigating the multitasking features of UNIX. This chapter deals with techniques for invoking programs and processes, using `exec`, `fork`, `wait`, and related system calls. The next chapter explains simple interprocess communication using pipes. Chapters 7 and 8 continue with more advanced interprocess communication mechanisms.

My presentation is organized with the goal of implementing a fairly complete command interpreter, or shell. We'll start with a limited shell that's barely usable. Then we'll keep adding features until we end up, in the next chapter, with a shell that can handle I/O redirection, pipelines, background processes, quoted arguments, and environment variables.

5.2 Environment

I'll begin with a discussion of the environment, which most UNIX users are already familiar with at the shell level. When a UNIX program is executed, it receives two collections of data from the process that invoked it: the *arguments* and the *environment.* To C programs, both are in the form of an array of character pointers, all but the last of which point to a NUL-terminated character string. The last pointer is `NULL`. A count of the number of arguments is also passed on. Other languages use a different interface, but we're concerned here only with C and C++.

A C or C++ program begins in one of two ways:[1]

1. Some implementations allow a third array-of-string argument that contains the environment, but this is both nonstandard and unnecessary.

main—C or C++ program entry point

```
int main(
    int argc,              /* argument count */
    char *argv[]           /* array of argument strings */
)

int main(void)
```

The count `argc` doesn't include the `NULL` pointer that terminates the `argv` array. If the program doesn't take arguments, the count and array can be omitted, as in the second form. I've already shown examples of both forms in this book.

In addition, the global variable `environ` points to the array of environment strings, also `NULL` terminated (there's no associated count variable):

environ—environment strings

```
extern char **environ;     /* environment array (not in any header) */
```

Each argument string can be anything at all, as long as it's NUL-terminated. Environment strings are more constrained. Each is in the form *name=value,* with the NUL byte after the value. Of course, the name can't contain a = character.

I'll get to how the environment is passed to `main` (causing `environ` to be set) in Section 5.3; here I just want to talk about retrieving values from and modifying `environ`.

One way to get at the environment is just to access `environ` directly, like this:

```
extern char **environ;

int main(void)
{
    int i;

    for (i = 0; environ[i] != NULL; i++)
        printf("%s\n", environ[i]);
    exit(EXIT_SUCCESS);
}
```

On Solaris here are some of the lines that this program printed (the rest were omitted to save space):

```
HOME=/home/marc
HZ=100
LC_COLLATE=en_US.ISO8859-1
LC_CTYPE=en_US.ISO8859-1
```

```
LC_MESSAGES=C
LC_MONETARY=en_US.ISO8859-1
LOGNAME=marc
MAIL=/var/mail/marc
```

Listing the whole environment is an unusual requirement; usually a program wants the value of a specific variable, and for that there's the Standard C function `getenv`:

getenv—get value of environment variable

```
#include <stdlib.h>

char *getenv(
    const char *var        /* variable to find */
);
/* Returns value or NULL if not found (errno not defined) */
```

`getenv` returns just the value part, to the right of the = sign, as in this example:

```
int main(void)
{
    char *s;

    s = getenv("LOGNAME");

    if (s == NULL)
        printf("variable not found\n");
    else
        printf("value is \"%s\"\n", s);
    exit(EXIT_SUCCESS);
}
```

which printed:

```
value is "marc"
```

Updating the environment is not as easy as reading it. Although the internal storage it occupies is the exclusive property of the process, and may be modified freely, there is no guarantee of extra room for any new variables or for longer values. So unless the update is trivial, a completely new environment must be created. If the pointer `environ` is then made to point to this new environment, it will be passed on to any programs subsequently invoked (as we shall see in Section 5.3), and it will be used by subsequent calls to `getenv`. Any updates have no effect on any other processes, including the shell (or whatever) that invoked the process doing the updating. Thus, if you want to modify the shell's environment, you have to do so with commands built into the shell.

Rather than messing with the environment directly, you can use some standard functions:

putenv—change or add to environment

```
#include <stdlib.h>

int putenv(
     char *string           /* string of form name=value */
);
/* Returns 0 on success or non-zero on error (sets errno) */
```

setenv—change or add to environment

```
#include <stdlib.h>

int setenv(
     const char *var,       /* variable to be changed or added */
     const char *val,       /* value */
     int overwrite          /* overwrite? */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

unsetenv—remove environment variable

```
#include <stdlib.h>

int unsetenv(
     const char *var        /* variable to be removed */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

All of the functions modify the storage pointed to by `environ`, perhaps setting `environ` to a new value if the array of pointers isn't long enough. If you've modified `environ` yourself or any parts of the environment it points to, the behavior of these functions is undefined, so decide whether you're going to do it yourself or use the functions but don't mix the two approaches.

`putenv` takes complete environment strings of the form name=value and makes a pointer in the `environ` array point to the storage you pass in, which then becomes part of the environment, so don't pass in any automatically allocated data (local, nonstatic variables), and don't modify the string after you've called `putenv`.

`setenv` is more sophisticated: It copies the variable name and value you pass in and allocates its own storage for it. If the variable already exists, its value is changed if the third `overwrite` argument is nonzero; otherwise, the old value

stays. If the variable doesn't already exist, it's added to the environment and the value of `overwrite` doesn't matter.

`unsetenv` comes with `setenv`, and it removes a variable and value from the environment. If your system doesn't have `unsetenv`, the best you can do to remove a variable from the environment is to set its value to the empty string, which may or may not be acceptable, depending on the application. (Some systems, such as Linux, FreeBSD, and Darwin, define `unsetenv` as a `void` function, so there's no error return.)

While the interfaces to these functions is standardized, their presence isn't required. FreeBSD, Linux, and Solaris all have `putenv`, and the first two have all three. SUS3 requires `setenv` and `unsetenv`, but as of this writing there aren't any SUS3 systems, so there's no simple, portable way to determine which are present. `setenv` and `unsetenv` are from BSD and should be present in all systems derived from it, including FreeBSD; `putenv` is from System V and should be present in all systems derived from it, including Solaris. Since the SUS recommends `setenv` and `unsetenv`, perhaps the best approach is to code with those functions and just include your own implementations for those systems that don't have them already.

It's easy to implement `setenv` and `unsetenv` if you don't care about memory leaks. The problem is that they may have to allocate or, in the case of `unsetenv`, orphan memory, but they can't free the memory because they can't assume anything about how it was allocated in the first place. That defect notwithstanding, here's our version of `setenv`:

```
int setenv(const char *var, const char *val, int overwrite)
{
    int i;
    size_t varlen;
    char **e;

    if (var == NULL || val == NULL || var[0] == '\0' ||
      strchr(var, '=') != NULL) {
        errno = EINVAL;
        return -1;
    }
    varlen = strlen(var);
    for (i = 0; environ[i] != NULL; i++)
        if (strncmp(environ[i], var, varlen) == 0 &&
          environ[i][varlen] == '=')
            break;
```

```
    if (environ[i] == NULL) {
        if ((e = malloc((i + 2) * sizeof(char *))) == NULL)
            return -1;
        memcpy(e, environ, i * sizeof(char *));
        /* possible memory leaks with old pointer array */
        environ = e;
        environ[i + 1] = NULL;
        return setnew(i, var, val);
    }
    else {
        if (overwrite) {
            if (strlen(&environ[i][varlen + 1]) >= strlen(val)) {
                strcpy(&environ[i][varlen + 1], val);
                return 0;
            }
            return setnew(i, var, val);
        }
        return 0;
    }
}
```

Comments about this function:

- The first thing we do is check the arguments, which is required by the SUS.
- Next we search the environment to see if the variable is already defined. Note that we have to look for a string that starts with the name followed by an = sign.
- If the variable is found, we exit if the `overwrite` argument is false. Otherwise, there are two cases: The new value fits, in which case we just copy it in, or it doesn't, in which case we call the function `setnew` (below) to put it in.
- If the variable is not found, we have to grow the array. We can't use `realloc` because we can't assume how the old memory was allocated, as we said. So we use `malloc` and copy the old contents in ourselves. Then we terminate the new array with a `NULL` pointer and call `setnew` to put the new entry in.

Here's `setnew`, which allocates space for a name, an = sign, and a value and stores it in the array:

```
static int setnew(int i, const char *var, const char *val)
{
    char *s;

    if ((s = malloc(strlen(var) + 1 + strlen(val) + 1)) == NULL)
        return -1;
    strcpy(s, var);
```

```
    strcat(s, "=");
    strcat(s, val);
    /* possible memory leak with old value of environ[i] */
    environ[i] = s;
    return 0;
}
```

Lastly, we have `unsetenv` that checks the argument, finds it, and, if it's there, slides the rest of the array down to effectively remove it. This also creates a possible memory leak, as we can't assume that the memory for the unset variable can be freed.

```
int unsetenv(const char *var)
{
    int i, found = -1;
    size_t varlen;

    if (var == NULL || var[0] == '\0' || strchr(var, '=') != NULL) {
        errno = EINVAL;
        return -1;
    }
    varlen = strlen(var);
    for (i = 0; environ[i] != NULL; i++)
        if (strncmp(environ[i], var, varlen) == 0 &&
          environ[i][varlen] == '=')
            found = i;
    if (found != -1)
        /* possible memory leak with old value of environ[found] */
        memmove(&environ[found], &environ[found + 1],
          (i - found) * sizeof(char *));
    return 0;
}
```

Note that we use `memmove` rather than `memcpy` because the former is guaranteed to work if the source and target memory areas overlap, as they certainly do in this case.

You may be wondering why we didn't use our "ec" error-checking macros in these functions. The reason is that we wanted their behavior to be as close to the standard functions as possible.

It's fairly straightforward to correct the memory-leak problems in `setenv` and `unsetenv`; see Exercise 5.1.

5.3 exec System Calls

It's impossible to understand the `exec` or `fork` system calls without fully understanding the distinction between a process and a program. If these terms are new to you, you may want to go back and review Section 1.1.2. If you're ready to proceed now, we'll summarize the distinction in one sentence: A process is an execution environment that consists of instruction, user-data, and system-data segments, as well as lots of other resources acquired at runtime, whereas a program is a file containing instructions and data that are used to initialize the instruction and user-data segments of a process.

The `exec` system calls reinitialize a process from a designated program; the program changes while the process remains. On the other hand, the `fork` system call (the subject of Section 5.5) creates a new process that is a clone of an existing one, by just copying over the instruction, user-data and system-data segments; the new process is not initialized from a program, so old and new processes execute the same instructions.

Individually, `fork` and `exec` are of limited use—mostly, they're used together. Keep this in mind as I present them separately, and don't be alarmed if you think they're useless—just try to understand what they do. In Section 5.4, when we use them together, you'll see that they are a powerful pair.

Aside from booting the UNIX kernel itself, `exec` is the only way programs get executed on UNIX. Not only does the shell use `exec` to execute our programs, but the shell and its ancestors were invoked by `exec`, too. And `fork` is the only way new processes get created.

Actually, there is no system call named "exec." The so-called "exec" system calls are a set of six, with names of the form `exec`*AB*, where *A* is either `l` or `v`, depending on whether the arguments are directly in the call (list) or in an array (vector), and *B* is either absent, a `p` to indicate that the `PATH` environment variable should be used to search for the program, or an `e` to indicate that a particular environment is to be used. (Oddly, you can't get both the `p` and `e` features in the same call.) Thus, the six names are `execl`, `execv`, `execlp`, `execvp`, `execle`, and `execve`.[2] We'll start with `execl`, and then do the other five.

2. The same features could be easily provided in two system calls instead of six, but it's a little late to change. See Exercise 5.6.

execl—execute file with argument list

```
#include <unistd.h>

int execl(
     const char *path,      /* program pathname */
     const char *arg0,      /* first arg (file name) */
     const char *arg1,      /* second arg (if needed) */
     ...,                   /* remaining args (if needed) */
     (char *)NULL           /* arg list terminator */
);
/* Returns -1 on error (sets errno) */
```

The argument `path` must name a program file that is executable by the effective user-ID (mode 755, say) and has the correct contents for executable programs. The process's instruction segment is overwritten by the instructions from the program, and the process's user-data segment is overwritten by the data from the program, with a re-initialized stack. Then the process executes the new program from the top (that is, its `main` function is called).

There can be no return from a successful `execl` because the return location is gone. An unsuccessful `execl` does return, with a value of –1, but there's no need to test for that value, for no other value is possible. The most common reasons for an unsuccessful `execl` is that the path doesn't exist or it isn't executable.

The arguments to `execl` that follow `path` are collected into an array of character pointers; the last argument, which must be `NULL`, stops the collection and terminates the array. The first argument, by convention, is the name of the program file (not the entire path). The new program may access these arguments through the familiar `argc` and `argv` arguments of `main`. The environment pointed to by `environ` is passed on, too, and it is accessible through the new program's `environ` pointer or with `getenv`, as explained in the previous section.

Since the process continues to live, and since its system-data segment is mostly undisturbed, almost all of the process's attributes are unchanged, including its process-ID, parent-process-ID, process-group-ID, session-ID, controlling terminal, real user-ID and group-ID, current and root directories, priority, accumulated execution times, and, usually, open file descriptors. It's simpler to list the principal attributes that *do* change, all for good reasons:

- If the process had arranged to catch any signals, they are reset to the default action, since the instructions designated to catch them are gone. Ignored or defaulted signals stay that way. (More about signals in Chapter 9.)
- If the set-user-ID or set-group-ID bit of the new program file is on, the effective user-ID or group-ID is changed to the owner-ID or group-ID of the file. There's no way to get the former effective IDs back, if they were themselves different from the real IDs.
- Any functions registered with `atexit` (Section 1.3.4) are unregistered, since their code is gone.
- Shared memory segments (Section 7.12) are detached (unmapped), as the attachment point is no longer valid.
- POSIX named semaphores (Section 7.10) are closed. System V semaphores (Section 7.9) are undisturbed.

For the complete list, see Appendix A. But you get the idea: If retaining an attribute or resource isn't meaningful with an entirely new program running, it's reset to its default or closed.

To show how `execl` is used, here's a rather contrived example:

```
void exectest(void)
{
    printf("The quick brown fox jumped over ");
    ec_neg1( execl("/bin/echo", "echo", "the", "lazy", "dogs.",
      (char *)NULL) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("exectest");
EC_CLEANUP_END
}
```

Here's the output we got when we called the function:

```
the lazy dogs.
```

What happened to the fox? Well, it turns out it wasn't quick enough: The standard I/O Library, of which `printf` is a part, buffers its output, and any partially full last buffer is automatically flushed when the process exits. But the process didn't exit before `execl` was called, and the buffer, being in the user-data segment, got overlaid before it could be flushed.

The problem can be fixed by either forcing unbuffered output, like this:

```
setbuf(stdout, NULL);
```

or by flushing the buffer just before calling `execl`, like this:

```
fflush(stdout);
```

As I've mentioned, open file descriptors normally stay open across an `execl`. If we don't want a file descriptor to stay open, we close it first, with `close`. But sometimes we can't. Suppose we are invoking a program that requires all file descriptors to be closed (an extremely unconventional requirement). We might therefore try this:

```
errno = 0;
ec_neg1( open_max = sysconf(_SC_OPEN_MAX) )
for (fd = 0; fd < open_max; fd++)
    (void)close(fd); /* ignore errors */
ec_neg1( execl(path, arg0, arg1, arg2, (char *)NULL) )
```

We call `sysconf` (Section 1.5.5) to get the maximum number of files that a process can have open, and the highest-numbered file descriptor is one less than that. We want so badly to close all file descriptors here that we don't even care if they're all open—we close them anyhow. It's one of the rare instances where ignoring system-call error returns is appropriate.

All is fine if the `execl` succeeds, but if it fails we'll never see an error message because file descriptor 2 has been closed! Here's where the close-on-exec flag that we encountered in Section 3.8.3 can be used. When set, with `fcntl`, the file descriptor is closed upon a successful `execl` but left alone otherwise. We can either set this flag for only the important file descriptors or just set it for all of them, like this:

```
for (fd = 0; fd < open_max; fd++) {
    ec_neg1( flags = fcntl(fd, F_GETFD) )
    ec_neg1( fcntl(fd, F_SETFD, flags | FD_CLOEXEC) )
}
ec_neg1( execl(path, arg0, arg1, arg2, (char *)NULL) )
```

A problem with closing or setting close-on-exec for all file descriptors is that there may be a thousand or more of them, and this is a lot of processing, mostly wasted. At the same time, it's wrong to leave any but the standard file descriptors open across an `exec`, unless the new program is expecting more than the standard three, which is unusual. So, in most cases, you need to keep track of what file descriptors your program has opened and close them explicitly before an `exec`.

Because the standard file descriptors are normally left open, close-on-exec is rarely used. An example where it is useful would be on a file descriptor open to a

log file, which you would want to keep open in case the `exec` fails and you want to log that fact.

The other five `exec` system calls provide three features not available with `execl`:

- Putting the arguments into an array instead of listing them explicitly. This is necessary when the number of arguments is unknown at compile time, as in programming a shell, for example.
- Searching for the program file using the value of the `PATH` environment variable, as a shell does.
- Manually passing an explicit environment pointer, instead of automatically using `environ`. Since a successful `exec` overwrites the existing environment anyhow, there's seldom an advantage to this feature.[3]

Here are the synopses for the `exec` variants I haven't shown yet:

execv—execute file with argument vector

```
#include <unistd.h>

int execv(
     const char *path,      /* program pathname */
     char *const argv[]     /* argument vector */
);
/* Returns -1 on error (sets errno) */
```

execlp—execute file with argument list and PATH search

```
#include <unistd.h>

int execlp(
     const char *file,      /* program file name */
     const char *arg0,      /* first arg (file name) */
     const char *arg1,      /* second arg (if needed) */
     ...,                   /* remaining args (if needed) */
     (char *)NULL           /* arg list terminator */
);
/* Returns -1 on error (sets errno) */
```

execvp—execute file with argument vector and PATH search

```
#include <unistd.h>

int execvp(
     const char *file,      /* program file name */
     char *const argv[]     /* argument vector */
);
/* Returns -1 on error (sets errno) */
```

3. But see Exercise 5.2.

execle—execute file with argument list and environment

```
#include <unistd.h>

int execle(
     const char *path,      /* program pathname */
     const char *arg0,      /* first arg (file name) */
     const char *arg1,      /* second arg (if needed) */
     ...,                   /* remaining args (if needed) */
     (char *)NULL,          /* arg list terminator */
     char *const envv[]     /* environment vector */
);
/* Returns -1 on error (sets errno) */
```

execve—execute file with argument vector and environment

```
#include <unistd.h>

int execve(
     const char *path,      /* program pathname */
     char *const argv[],    /* argument vector */
     char *const envv[]     /* environment vector */
);
/* Returns -1 on error (sets errno) */
```

Note that the `argv` argument as used here is identical in layout to the `argv` argument of a `main` function. Don't forget that the last pointer has to be `NULL`.

If a `file` argument to `execlp` or `execvp` has no slashes, the strings listed in the value of the `PATH` variable are prepended to it one by one until an ordinary file with the resulting path name that has execute permission is located. If this file contains a program (as indicated by a code number in its first word), it is executed. If not, it is assumed to be a script file; usually, to run it a shell is executed with the path as its first argument.

For example, if the `file` argument is `echo`, and a search of the `PATH` strings finds an executable whose path is /bin/echo, then that file is executed as a program because it contains binary instructions in the right format. But if the `file` argument is, say, `myscript`, and a search turns up /home/marc/myscript which is not a binary executable, `execlp` or `execvp` instead executes the equivalent of:

```
sh /home/marc/myscript arg1 arg2 …
```

It's up to the implementation as to what shell it uses, but it must be standards-conforming.

There is one other universally supported, but nonstandard, convention: If the first line of the script is of the form

```
#! pathname [arg]
```

the interpreter specified by `pathname` is executed, instead of the shell, with the pathname of the script file as its first argument. If there's an optional `arg` on the `#!` line, it becomes the first argument to the program, and the script pathname becomes the second. The interpreter has to bypass the `#!` line, which is why most UNIX scripting languages use `#` to start a comment.

For example, you can put a script like this in the file `mycmd`:

```
#! /usr/bin/python2.2
print "Hello World!"
```

make it executable, and then execute it like this:

```
$ mycmd
Hello World!
$
```

On nearly all systems, the work to deal with the `#!` line is done by `execvp`, not by the shell itself.

If the `PATH` search turns up nothing executable, the `exec` fails. If the `file` argument has slashes in it, no search is done—the path is assumed to be complete. It still may be a script, however.

It's possible to program `execvp` and `execlp` as library functions that call on `execv` or `execl`, and we can learn a lot by doing so. I'll show a version of `execvp` that's close to the standard version, but which uses the "ec" error-checking macros.

Recall that for each path found by searching `PATH`, `execvp` first has to try to execute it as a binary executable and then again as a shell script. Here's a function that does that much (omitting the `#!` feature):

```
int exec_path(const char *path, char *const argv[], char *newargv[])
{
    int i;

    execv(path, argv);
    if (errno == ENOEXEC) {
        newargv[0] = argv[0];
        newargv[1] = (char *)path;
        i = 1;
        do {
            newargv[i + 1] = argv[i];
        } while (argv[i++] != NULL);
        return execv("/bin/sh", (char *const *)newargv);
    }
```

```
    return -1;
}
```

`exec_path` returns –1 with `errno` set if it fails to execute the path one way or another.

Our version of `execvp`, called `execvp2`, has to try each of the paths in `PATH` in turn, calling `exec_path` each time, unless the passed-in file contains a slash or `PATH` is nonexistent, in which case it just uses it as is:

```
int execvp2(const char *file, char *const argv[])
{
    char *s, *pathseq = NULL, *path = NULL, **newargv = NULL;
    int argc;

    for (argc = 0; argv[argc] != NULL; argc++)
        ;
    /* If shell script, we'll need room for one additional arg and NULL. */
    ec_null( newargv = malloc((argc + 2) * sizeof(char *)) )
    s = getenv("PATH");
    if (strchr(file, '/') != NULL || s == NULL || s[0] == '\0')
        ec_neg1( exec_path(file, argv, newargv) )
    ec_null( pathseq = strdup(s) )
    /* Following line usually allocates too much */
    ec_null( path = malloc(strlen(file) + strlen(pathseq) + 2) )
    while ((s = strtok(pathseq, ":")) != NULL) {
        pathseq = NULL; /* tell strtok to keep going */
        if (s[0] == '\0')
            s = ".";
        strcpy(path, s);
        strcat(path, "/");
        strcat(path, file);
        exec_path(path, argv, newargv);
    }
    errno = ENOENT;
    EC_FAIL

EC_CLEANUP_BGN
    free(pathseq);
    free(path);
    free(newargv);
    return -1;
EC_CLEANUP_END
}
```

Some comments about the memory allocations follow:

- It's not necessary or even possible to free up allocated memory if an `exec` is successful, as all user data will be overwritten.
- Taking the length of `PATH` plus the length of the passed-in file name plus two more bytes for a slash and a NUL is definitely enough for every path we'll synthesize.
- If we're going to try executing a shell script, we need an argument vector with one more slot than the one passed in, so we count the number of arguments passed in (in the `for` loop) and then allocate a vector with two more than that many slots (one for the additional argument—the pathname for the shell to execute—and one for the `NULL` pointer at the end).
- We use `strdup` to allocate a new string to hold the value of `PATH` because `strtok` will write into it. The pointer we got directly from `getenv` points into the environment, and we don't want to modify those strings directly, in case `execvp2` fails. (Setting the first argument of `strtok` to `NULL` is something you have to do when calling that function—that's how it knows to continue scanning the original string.)

As we've said, normally `exec` is paired with `fork`, but occasionally it's useful by itself. Sometimes a large program is split into phases, and `exec` brings in the next phase. But since *all* instructions and user-data are overlaid, the phases would have to be quite independent for this to be workable—data can be passed only through arguments, the environment, or files. So this application is rare but not unknown. More commonly, `exec` is used by itself when we need to do a very small amount of preliminary work before invoking a command. An example is the `nohup` command, which suppresses hangup signals before calling `execvp` to invoke the user's command. Another example is the `nice` command, which changes a command's priority (see Section 5.15 for an example).

5.4 Implementing a Shell (Version 1)

We now know enough system calls to write a shell of our own, although not a very good one. Its chief deficiency is that it has to commit suicide in order to execute a command, since there is no return from `exec`. Perversely, if you type in bad commands, you can keep running. While we're at it, we'll implement two built-in commands to modify and access the environment: assignment (e.g., `BOOK=/usr/marc/book`) and `set`, which prints the environment.[4]

4. There's no `export` command; the whole environment is passed to executed commands.

The first task is to break up a command line into arguments. For simplicity, we'll do without quoted arguments. Moreover, since this shell can't handle background processes, sequential execution, or piping, we don't have to worry about the special characters `&`, `;`, and `|`. We'll leave out redirection (`>`, `>>`, and `<`) also. Therefore, a command line is just a series of words separated by blanks or tabs. We have to gather them up and put them into an `argv` array:

```
#define MAXLINE 200

static bool getargs(int *argcp, char *argv[], int max, bool *eofp)
{
    static char cmd[MAXLINE];
    char *cmdp;
    int i;

    *eofp = false;
    if (fgets(cmd, sizeof(cmd), stdin) == NULL) {
        if (ferror(stdin))
            EC_FAIL
        *eofp = true;
        return false;
    }
    if (strchr(cmd, '\n') == NULL) {
        /* eat up rest of line */
        while (true) {
            switch (getchar()) {
            case '\n':
                break;
            case EOF:
                if (ferror(stdin))
                    EC_FAIL
            default:
                continue;
            }
            break;
        }
        printf("Line too long -- command ignored\n");
        return false;
    }
    cmdp = cmd;
    for (i = 0; i < max; i++) {
        if ((argv[i] = strtok(cmdp, " \t\n")) == NULL)
            break;
        cmdp = NULL; /* tell strtok to keep going */
    }
    if (i >= max) {
        printf("Too many args -- command ignored\n");
        return false;
    }
```

```
    *argcp = i;
    return true;

EC_CLEANUP_BGN
    EC_FLUSH("getargs")
    return false;
EC_CLEANUP_END
}
```

Comments:

- `getargs` returns `true` if it parsed the arguments OK, and otherwise `false`.
- It uses the standard C function `fgets` to read a line of input. The function is safe enough if a long line is typed, since it won't overrun the buffer we pass in. But in that case we don't want to leave the unread characters to be read later, as they would then be an unintended, and potentially damaging, shell command in their own right. So in that case (no newline terminator) we read in and throw away the rest of the line before printing an error message.
- Then we use `strtok` to break up the line into its arguments. (We also used `strtok` in `execvp2`, in the previous section.)
- `getargs` displays its own error messages with `EC_FLUSH` in case of a genuine error from one of the library functions it calls, and otherwise—such as in the case of a line that's too long—prints a message for the user.

Next, we need functions to handle the built-in commands, assignment and `set`. These are easy, since we can use the environment-manipulation techniques from Section 5.2:

```
extern char **environ;

void set(int argc, char *argv[])
{
    int i;

    if (argc != 1)
        printf("Extra args\n");
    else
        for (i = 0; environ[i] != NULL; i++)
            printf("%s\n", environ[i]);
}

void asg(int argc, char *argv[])
{
    char *name, *val;
```

```
    if (argc != 1)
        printf("Extra args\n");
    else {
        name = strtok(argv[0], "=");
        val = strtok(NULL, ""); /* get all that's left */
        if (name == NULL || val == NULL)
            printf("Bad command\n");
        else
            ec_neg1( setenv(name, val, true) )
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("asg")
EC_CLEANUP_END
}
```

Next, we complete the program with a `main` function that prints the prompt character (we use `@`), gets the arguments, checks to see if the command is built in, and if it's not, tries to execute it:

```
#define MAXARG 20

int main(void)
{
    char *argv[MAXARG];
    int argc;
    bool eof;

    while (true) {
        printf("@ ");
        if (getargs(&argc, argv, MAXARG, &eof) && argc > 0) {
            if (strchr(argv[0], '=') != NULL)
                asg(argc, argv);
            else if (strcmp(argv[0], "set") == 0)
                set(argc, argv);
            else
                execute(argc, argv);
        }
        if (eof)
            exit(EXIT_SUCCESS);

    }
}

static void execute(int argc, char *argv[])
{
    execvp(argv[0], argv);
    printf("Can't execute\n");
}
```

Note that we loop back to print another prompt only if `execvp` fails. That's why we're unlikely to find this shell useful. Here's a sample session anyhow (the listing of the environment has been abbreviated to save space):

```
$ sh0
@ set
SSH_CLIENT=192.168.0.1 4971 22
USER=marc
MAIL=/var/mail/marc
EDITOR=vi
@ LASTNAME=Rochkind
@ set
SSH_CLIENT=192.168.0.1 4971 22
USER=marc
MAIL=/var/mail/marc
EDITOR=vi
LASTNAME=Rochkind
@ echo Hello World!
Hello World!
$
```

Note the dollar-sign prompt at the end—after executing `echo`, our shell has exited. Actually, it was `echo` that exited, as it was the program that replaced the shell. Clearly, it would be better for a shell to execute commands in processes of their own, and that's exactly where we're headed.

5.5 `fork` System Call

`fork` is somewhat the opposite of `exec`: It creates a new process, but it does not initialize it from a new program. Instead, the new process's instruction, user-data, and system-data segments are almost exact copies of the old process's.

fork—create new process

```
#include <unistd.h>

pid_t fork(void);

/* Returns child process-ID and 0 on success or -1 on error (sets errno) */
```

After `fork` returns, *both* processes (parent and child) receive the return. The return value is different, however, which is crucial, because this allows their subsequent actions to differ. Usually, the child does an `exec`, and the parent either waits for the child to terminate or goes off to do something else.

The child receives a 0 return value from `fork`; the parent receives the process-ID of the child. As usual, a return of –1 indicates an error, but since `fork` has no

arguments, the caller could not have done anything wrong. The only cause of an error is resource exhaustion, such as insufficient swap space or too many processes already in execution. Rather than give up, the parent may want to wait a while (with `sleep`, say) and try again, although this is not what shells typically do—they print a message and reprompt.

Recall that a program invoked by an `exec` system call retains many attributes because the system-data segment was left mostly alone. Similarly, a child created by `fork` inherits most attributes from its parent because its system-data segment is copied from its parent. It is this inheritance that allows a user to set certain attributes from the shell, such as current directory, effective user-ID, and priority, that then apply to each command subsequently invoked. One can imagine these attributes belonging to the "immediate family," although that's not a formally defined UNIX term.

Only a few attributes are not inherited:

- Obviously, the child's process-ID and parent-process-ID are different, since it's a different process.
- If the parent was running multiple threads (Section 5.17), only the one that executed the `fork` exists in the child. All the threads remain intact in the parent, however.
- The child gets duplicates of the parent's open file descriptors. Each is opened to the same file, and the file pointer has the same value. The open file *description* (Section 2.2.3) and, hence, the file offset, is shared. If the child changes it with `lseek`, then the parent's next `read` or `write` will be at the new location. The file *descriptor* itself, however, is distinct: If the child closes it, the parent's copy is undisturbed.
- The child's accumulated execution times are reset to zero because it's at the beginning of its life.

(These are the main ones—see Appendix A for the complete list.)

Here's a simple example to show the effect of a `fork`:

```
void forktest(void)
{
    int pid;

    printf("Start of test\n");
    pid = fork();
    printf("Returned %d\n", pid);
}
```

The output:

```
$ forktest
Start of test
Returned 98657
Returned 0
$
```

In this case the parent executed its `printf` before the child, but in general you can't depend on which comes first because the processes are independently scheduled. If it matters, you'll have to synchronize the two yourself, as explained in Section 9.2.3. As shown there, you can do it with a pipe or, less easily, with a signal.

Let's run `forktest` again, but this time with the standard output redirected to a file:

```
$ forktest >tmp
$ cat tmp
Start of test
Returned 56807
Start of test
Returned 0
$
```

This time "Start of test" got written twice! Can you figure out why? (Spoiler in next paragraph.)

What happened was that `printf` buffered its output (as explained in Section 2.12) and the child inherited the unflushed buffer, along with everything else. When the child exited, it flushed the buffer, and so did the parent. Before, when the output wasn't redirected, `printf` didn't buffer because it knew that the standard output was a terminal device and that it should behave more interactively. I'll discuss how to control the side-effects of exiting in Section 5.7.

`fork` and `exec` are a perfect match because the child process created by `fork`, as it is a near clone of the parent, isn't usually useful. It's ideal, though, for being overlaid by a new program, which is precisely what `exec` does. We'll see that in the next section, as we improve on our shell.

The cost of a `fork` is potentially enormous. It goes to all the trouble of cloning the parent, perhaps copying a large data segment (the instruction segment is usually read-only and can be shared) only to execute a few hundred instructions before reinitializing the code and data segments. This seems wasteful and it is. A

clever solution is used in virtual-memory versions of UNIX, in which copying is particularly expensive, called *copy-on-write.* It works this way: On a `fork`, the parent and child share the data segment, which consists of a collection of *pages.* As long as a page stays unmodified, this shortcut works OK. Then as soon as parent or child attempts to write on a page, just that page is copied to give each its own version. This is transparent and efficient with the right kind of dynamic-address-translation hardware, which almost all modern-day computers have. Since the `exec` follows very quickly in almost all cases, very few pages will have to be copied. But even if they all have to be copied, we are no worse off—in fact, we're better off, since we were able to start running the child earlier. Remember that a copy-on-write scheme is internal to the kernel; it does not change the semantics of `fork`, and, apart from better performance, users are unaware of it.

Even more efficient than copy-on-write, but not as transparent, is a variation on `fork` called `vfork`:

vfork—create new process; share memory (*obsolete*)

```
#include <unistd.h>

pid_t vfork(void);

/* Returns child process-ID and 0 on success or -1 on error (sets errno) */
```

`vfork` behaves exactly like `fork` insofar as creating the child process is concerned, but parent and child share the same data segment. There's no copy-on-write—they both read and write the same variables, so havoc ensues unless the child immediately exits or does an `exec`, which is what normally happens anyway (see the example function `execute2` in the next section).[5] In early versions of BSD UNIX, `vfork` was important; however, its performance advantages over `fork` are no longer substantial and its use is dangerous, so you should not use it, which is why it's marked "obsolete" in the synopsis.

A more recent attempt to improve the efficiency of `fork` and `exec` is `posix_spawn`, part of a 1999 POSIX update. Because its end result is a child process running a different program, it avoids the problems of `vfork`. It doesn't offer all the flexibility of a separate `fork` and `exec`, but it does handle the most common cases (e.g., duplicating a file descriptor). The real purpose of

5. Two processes concurrently reading and writing the same variables is similar to threads, with similar potential dangers. Section 5.17 has more.

`posix_spawn` is to provide an efficient way to invoke a program as a child process in a realtime system when swapping would be too slow and the hardware lacks dynamic address translation. It's possible to implement `posix_spawn` as a library function that uses `fork` and `exec` (Exercise 5.8), although in a realtime system of the sort its designers envisioned `posix_spawn` would have to be implemented as a system call.

It's a good time to mention the Standard C function `system`:

system—run command

```
#include <stdlib.h>

int system(
    const char *command       /* command */
);
/* Returns exit status or -1 on error (sets errno) */
```

This function does a `fork` followed by an `exec` of the shell, passing the `command` argument as a shell command line. It's a handy way to execute a command line from within a program, but it doesn't provide the efficiency (the shell is always invoked) or flexibility of a separate `fork` followed by an `exec`, or of `posix_spawn`.

5.6 Implementing a Shell (Version 2)

Now let's put `fork` and `execvp` together to make our shell from the previous section far more useful—it won't just exit after running a command but will actually prompt for another one! We replace the function `execute` with `execute2`:

```
static void execute2(int argc, char *argv[])
{
    pid_t pid;

    switch (pid = fork()) {
    case -1: /* parent (error) */
        EC_FAIL
    case 0: /* child */
        execvp(argv[0], argv);
        EC_FAIL
    default: /* parent */
        ec_neg1( wait(NULL) )
    }
    return;
```

```
EC_CLEANUP_BGN
    EC_FLUSH("execute2")
    if (pid == 0)
        _exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

I'll present the details of `wait` fully in Section 5.8*;* for now, you just need to know that it waits for the child process to terminate before returning.

This function treats an error from `fork` and `wait` differently from an error from `execvp`. If `fork` or `wait` returns –1, we're in the parent, so we just print the error information (with `EC_FLUSH`) and return. The caller (`main` from the previous section) will then prompt the user again, which is what we want—a shell shouldn't terminate just because there's been an error. But if `execvp` returns, we are in the *child* process, not the parent, and we had better exit. Otherwise, the child would keep running and we would have *two* shells prompting for commands. If we actually tried to type a command, the two processes would compete for characters and each would get some of them, since a single character can only be read once. This could have serious consequences if, for example, we typed `rm t*` and one of the shells read `rm *` while the other read `t`.

We called `_exit` instead of `exit` for reasons I'm that going to explain next.

5.7 `exit` System Calls and Process Termination

There are exactly four ways that a process can terminate:

1. Calling `exit`. Returning a value from `main` is the same as calling `exit` with that value as an argument, and returning without a value is the same as returning 0.
2. Calling `_exit` or its equivalent, `_Exit`, two variants of `exit` that will be explained shortly.
3. Receiving a terminating signal.
4. A system crash, caused by anything from a power failure to a bug in the OS.

This section is concerned with the first two, Chapter 9 deals with signals, and there's really nothing to say about crashes, upon which processes just stop.[6]

6. Some computers send a signal to processes when power fails, but that just changes it to the third case.

The differences between the three variants of `exit` are:

- `_exit` and `_Exit` behave identically in all respects, although technically `_exit` is from UNIX and `_Exit` is from Standard C. From now on, I'll refer only to `_exit`, and you can assume everything I say about it applies to `_Exit` as well.
- `exit` (without the underscore) is also from Standard C. It does everything that `_exit` does and also some higher-level cleanup, including at least calling functions registered with `atexit` (Section 1.3.4) and flushing standard I/O buffers, as though `fflush` or `fclose` were called. (Whether `_exit` flushes the buffers is implementation defined, so don't count on it.)

As `exit` is a superset of `_exit`, everything I say here about the latter applies also to the former, so I'll mostly just talk about `_exit`.

Usually, you call `_exit` in a child process that hasn't done an `exec`, rather than `exit`, as I showed in `execute2` in the previous section, because whatever cleanup processing the child inherited usually should be done only once. But this isn't always true, so you'll have to examine each case to decide what's appropriate. In the most common case, when the child is overwritten by an `exec`, a program starts fresh, so whatever cleanup it does when it exits is OK. Any buffers or `atexit` functions from the parent are long gone.

If you're using the "ec" macros for error checking, as I do in this book, remember that the automatic display of an error message and the function trace-back are done in a function registered with `atexit` that won't be called if you terminate with `_exit` (Section 1.4.2). In `execute2` in the previous section we took care of this by using the `EC_FLUSH` macro in both parent and child, although only the child calls `_exit`.

Here are the synopses for the `exit` functions. Note that there are two different include files involved:

_exit—terminate process without cleanup

```
#include <unistd.h>

void _exit(
    int status          /* exit status */
);
/* Does not return */
```

_Exit—terminate process without cleanup

```
#include <stdlib.h>

void _Exit(
    int status              /* exit status */
);
/* Does not return */
```

exit—terminate process with cleanup

```
#include <stdlib.h>

void exit(
    int status              /* exit status */
);
/* Does not return */
```

`_exit` and the other two variants terminate the process that issued it, with a status code equal to the rightmost (least significant) byte of `status`. There are lots of additional side-effects; the principal ones are listed here, and you can find the complete list in [SUS2002]. Actually, these side-effects apply to process termination in general (except for a crash):

- All open file descriptors are closed.
- As explained in Section 4.3, if the process was the controlling process (the session leader), the session loses its controlling terminal. Also, every process in the session gets a hangup (`SIGHUP`) signal, which, if not caught or ignored, causes them to terminate as well.
- The parent process is notified of the exit status in a manner explained in the next section.
- Child processes aren't affected in any way directly, except that their new parent is a special system process, and if they execute the `getppid` (Section 5.13) system call, they'll get the process-ID of that system process (usually 1).

The exiting process's parent receives its status code via one of the `wait` system calls, which are in the next section. The status code is a number from 0 through 255. By convention, 0 means successful termination, and nonzero means abnormal termination of some sort, as defined by the program doing the exiting. Two standard macros are defined, which I've already used in many examples in this book: `EXIT_SUCCESS` is defined as 0, and `EXIT_FAILURE` is defined as some nonzero value, typically 1. Using the macros instead of integers like 0 and 1 makes programs a bit more readable, which is why we do it that way.

Once a process terminates with an `exit` variant or via a signal, it stops executing but doesn't go away completely until its exit status is reported to its parent, unless the system has determined (in a manner described in the next section) that its parent isn't interested in the status. A terminated process that hasn't yet reported is called a *zombie*.[7]

5.8 `wait`, `waitpid`, and `waitid` System Calls

`wait`, `waitpid`, or `waitid` waits for a child process to change state (stop, continue, or terminate) and then retrieves that process's status. How a process terminates was explained in the previous section; stopped and continued processes were explained in Section 4.3. I'll start with `waitpid` and then cover the other two variants.

Several of the features described in this and the next section are only in X/Open-conforming systems and, as of this writing, are not yet in FreeBSD or Linux; they're marked with the notation "[X/Open]." (X/Open was discussed in Sections 1.2 and 1.5.1; although Linux as of this writing claims to be X/Open when we test it with feature-test macros, it doesn't have the X/Open functionality described in this section.)

waitpid—wait for child process to change state

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

pid_t waitpid(
     pid_t pid,             /* process or process-group ID */
     int *statusp,          /* pointer to status or NULL */
     int options            /* options (see below) */
);
/* Returns process ID or 0 on success or -1 on error (sets errno) */
```

The `pid` argument can be used for one of four things:

`>0` wait for the child process with that process ID

`-1` wait for any child process

7. A dictionary definition of *zombie* that fits very well is "a person of the lowest order of intelligence suggestive of the so-called walking dead." (*Webster's New Collegiate Dictionary.* 6th ed. [Springfield, Mass.: G. & C. Merriam Co., 1960].)

`0` wait for any child process in the same process group as the calling process

`<-1` wait for any child process in the process group whose process-group ID is `-pid`

Process groups were explained in Section 4.3. Typically, a shell that runs each pipeline in its own process group will set `pid` to the negative of that process group's ID when waiting for the pipeline to complete so that any process in the pipeline will cause `waitpid` to return.

When `waitpid` returns because a child matching the `pid` argument changed state, the child's process ID is returned. Zero is returned in one case that will be explained below, under the `WNOHANG` option.

Only direct children—those created with `fork`—can be waited for. Grandchildren may not be, even if their parent (the direct child) has already terminated. As explained in the previous section, orphaned processes are inherited by a special system process, not by the grandparent.

Normally, it's important for a process to wait for every child it creates—otherwise terminated children remain in the system as zombies until the parent terminates, at which point the system process that inherits them will do the wait and clean them out for good. As some processes don't terminate for a long time (months, sometimes), keeping zombies around for that long really clogs up the system tables. If waiting is too troublesome, a process can use signals to eliminate zombies, as explained in the next section.

A child that changes state, called *waitable*, can cause at most one return from `waitpid`. In other words, a waitable child that's waited for is no longer waitable. This means that if one section of the program gets the status and finds out it wasn't from the process it was expecting, there's no way to stuff the result back into the system so some other `waitpid` will get the status for that process later. (The call `waitid` can do this, though—see below.)

If `waitpid` is executed and there's a waitable child that meets the `pid` specification, `waitpid` returns immediately. If there is such a child but it hasn't yet changed state, `waitpid` blocks until there is a suitable waitable child. If there is no child matching `pid` at all, `waitpid` returns `-1` and sets `errno` to `ECHILD`. There might be no child because the `pid` is simply wrong or because the child has already been waited for, in which case it's no longer waitable (but, again, see `waitid`).

If `statusp` is non-`NULL`, the integer it points to is set to the status of the child. This is a combination of the child's argument to `_exit` or `exit` (if that's how it terminated) and a code that indicates how it terminated or stopped. Macros are supplied for interpreting the integer:

`WIFEXITED(status)`	`true` if the child terminated normally (with `_exit` or `exit`[8])
`WEXITSTATUS(status)`	if `WIFEXITED`, the low-order 8 bits of the argument to `_exit` or `exit`
`WIFSIGNALED(status)`	`true` if the child terminated abnormally (because of a signal)
`WTERMSIG(status)`	if `WIFSIGNALED`, the number of the signal that caused termination
`WIFSTOPPED(status)`	`true` if the child stopped; possible only if the `WUNTRACED` option is set (see below)
`WSTOPSIG(status)`	if `WIFSTOPPED`, the number of the signal that caused the child to stop
`WIFCONTINUED(status)`	true if the process was continued; possible only if the `WCONTINUED` option is set [X/Open]
`WCOREDUMP(status)`	true if a memory-dump file (called a "core dump" in UNIX) was produced; useful sometimes for post-mortem analysis (macro is nonstandard, but commonly implemented)

The last argument, `options`, is one or more of the flags ORed together:

`WCONTINUED`	Report on continued children, in addition to those that terminated. [X/Open]
`WNOHANG`	Do not wait for a child if status is not immediately available; return 0 instead of a process ID.
`WUNTRACED`	Report on stopped children, in addition to those that terminated.[9]

8. Remember that `_Exit` is the same as `_exit`, and returning from `main` is the same as calling `exit`.
9. Should be called `WSTOPPED`, but called `WUNTRACED` for historical reasons.

Here are some example usages of `waitpid`:

```
/* Wait for child pid to terminate and get its status. */
ec_neg1( waitpid(pid, &status, 0) )

/* Wait for any child to terminate, without getting its status. */
ec_neg1( pid = waitpid(-1, NULL, 0) )

/* Report on any child in process group pgid to terminate or stop,
   and get its status. Don't wait if no status is available immediately. */
ec_neg1( pid = waitpid(-pgid, &status, WNOHANG | WUNTRACED) )
```

Here's an example program that creates three child processes, each of which terminates differently. For each termination, the status is reported.

```
int main(void)
{
    pid_t pid;

    /* Case 1: Explicit call to _exit */
    if (fork() == 0) /* child */
        _exit(123);
    /* parent */
    ec_false( wait_and_display() )

    /* Case 2: Termination by kernel */
    if (fork() == 0) { /* child */
        int a, b = 0;

        a = 1 / b;
        _exit(EXIT_SUCCESS);
    }
    /* parent */
    ec_false( wait_and_display() )

    /* Case 3: External signal */
    if ((pid = fork()) == 0) { /* child */
        sleep(100);
        _exit(EXIT_SUCCESS);
    }
    /* parent */
    ec_neg1( kill(pid, SIGHUP) )
    ec_false( wait_and_display() )

    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static bool wait_and_display(void)
{
    pid_t wpid;
    int status;

    ec_neg1( wpid = waitpid(-1, &status, 0) )
    display_status(wpid, status);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

void display_status(pid_t pid, int status)
{
    if (pid != 0)
        printf("Process %ld: ", (long)pid);
    if (WIFEXITED(status))
        printf("Exit value %d\n", WEXITSTATUS(status));
    else {
        char *desc;
        char *signame = get_macrostr("signal", WTERMSIG(status), &desc);
        if (desc[0] == '?')
            desc = signame;
        if (signame[0] == '?')
            printf("Signal #%d", WTERMSIG(status));
        else
            printf("%s", desc);
        if (WCOREDUMP(status))
            printf(" - core dumped");
        if (WIFSTOPPED(status))
            printf(" (stopped)");
#if defined(_XOPEN_UNIX) && !defined(LINUX)
        else if (WIFCONTINUED(status))
            printf(" (continued)");
#endif
        printf("\n");
    }
}
```

As we mentioned, the line

```
#if defined(_XOPEN_UNIX) && !defined(LINUX)
```

tests separately for Linux because it claims falsely to be X/Open conforming.

The function `get_macrostr` provides a printable version of a signal number (among other things), and it works a lot like `errsymbol` in Section 1.4.1. The code is on the Web site [AUP2003] if you're interested.

This was the output:

```
Process 9585: Exit value 123
Process 9586: Erroneous arithmetic operation - core dumped
Process 9587: Hangup
```

As the output shows, in case 1, the child terminated itself with a call to `_exit`, passing the exit value 123. In case 2, the child was terminated by a `SIGFPE` (floating-point error) signal when it tried to divide by zero. In case 3, the sleeping child was killed by a `SIGHUP` signal sent from its parent. (The `kill` system call, which sends signals, is in Section 9.1.9.)

We're going to use function `display_status` in the next version of our shell, later in this chapter.

A second variant, the `wait` system call, is shorthand for `waitpid` with a `pid` of `-1` and no `options`:

wait—wait for child process to terminate

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

pid_t wait(
    int *statusp,          /* pointer to status or NULL */
);
/* Returns process ID or -1 on error (sets errno) */
```

Because `wait` and `waitpid` with `pid` set to `-1` wait for any child, they're often inappropriate when a function that's part of a larger program creates a child that it wants to wait for. It might accidentally get a return for some other child entirely, and as a child can be waited for only once, that would deprive some other part of the program of the ability to wait for that child, causing potential confusion. It's much better to use `waitpid` to wait for a specific child, or at least a member of a process group, and for that reason `wait` is seldom used outside of simple programs. It should never be used in library functions that create child processes.

But suppose a process has two children and can't predict which will terminate first. If all it cares about is waiting for both of them to terminate, it can simply wait for either one of them and then, when that call to `waitpid` returns, wait for the other. Or, the parent can proceed with its own work and periodically issue a `waitpid` call for each process with the `WNOHANG` option. But, as we said, it should not issue a blanket `waitpid` call (`pid` argument of `-1`) unless it can guarantee that there are no other children. Generally, in big, complicated programs where parts came from different development groups (e.g., an imaging library, a database interface), there can be no such guarantee. It would be nice if one could pass an array of process IDs to some `wait` variant, but one can't.

The third variant, `waitid`, entered UNIX with SUS1, and so as of this writing isn't in Linux or FreeBSD.[10] It offers one important new feature: You can get the status of a process while keeping it waitable, so no harm is done if you query the wrong process. In addition, it supplies more information than `waitpid` or `wait`. Here's the synopsis:

waitid—wait for child process to change state [X/Open]

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

int waitid(
    idtype_t idtype,        /* interpretation of id */
    id_t id,                /* process or process-group ID */
    siginfo_t *infop,       /* returned info */
    int options             /* options */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

siginfo_t—structure for waitid[11]

```
typedef struct {            /* only waitid-relevant members shown */
    int si_code;            /* code (see below) */
    pid_t si_pid;           /* child process ID */
    uid_t si_uid;           /* real user ID of child process */
    int si_status;          /* exit value or signal */
} siginfo_t;
```

The first argument to `waitid`, `idtype`, is one of:

`P_PID` wait for child with process ID of `id` (like `waitpid` with `pid` > 0)

10. It seems to be only partially implemented on my version of Linux (it seems to work, but only the options for `waitpid` are defined, and there's no `man` page). You'll have to check your own version.
11. This structure has been augmented by the Realtime Signals Extension; see Section 9.5.

`P_PGID` wait for any child in process group `id` (like `waitpid` with `pid` < –1)

`P_ALL` wait for any child (like `waitpid` with `pid` == –1); `id` is ignored

There's no direct equivalent to `waitpid` with `pid` == 0, but the same thing can be done with `P_PID` and `id` equal to the caller's process-group ID.

The `options` argument, as with `waitpid`, is one or more flags ORed together:

`WEXITED` Report on processes that have exited. (Always the case with `waitpid`, which has no such flag.)

`WSTOPPED` Report on stopped children (similar to `waitpid` with the `WUNTRACED` flag).

`WCONTINUED` Report on continued children (same flag as for `waitpid`).

`WNOHANG` Do not wait for a child if status is not immediately available; return 0 (same flag as for `waitpid`).

`WNOWAIT` Leave the reported-on process waitable, so a subsequent `waitid` (or other variant) call can be used on it.

Through its `infop` argument, `waitid` provides more information than does `waitpid`. This argument points to a `siginfo_t` structure; the members relevant to its use with `waitid` are shown in the synopsis. On return, the child's process ID is in the `si_pid` member. You check `si_code` to see why the child terminated; the most commonly occurring codes are:

`CLD_EXITED` Exited with `_exit` or `exit`

`CLD_KILLED` Terminated abnormally (via a signal)

`CLD_DUMPED` Terminated abnormally and created a memory-dump file (called a "core" file in UNIX)

`CLD_STOPPED` Stopped

`CLD_CONTINUED` Continued

If the code is `CLD_EXITED`, the `si_status` member holds the number passed to `_exit` or `exit`. Otherwise, the state change was caused by a signal, and `si_status` is the signal number.

Here are the `wait_and_display` and `display_status` functions from the previous `waitpid` example rewritten for `waitid`:

```
static bool wait_and_display(void)
{
    siginfo_t info;

    ec_neg1( waitid(P_ALL, 0, &info, WEXITED) )
    display_status(&info);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

void display_status(siginfo_t *infop)
{
    printf("Process %ld terminated:\n", (long)infop->si_pid);
    printf("\tcode =  %s\n",
      get_macrostr("sigchld-code", infop->si_code, NULL));
    if (infop->si_code == CLD_EXITED)
        printf("\texit value =  %d\n", infop->si_status);
    else
        printf("\tsignal = %s\n",
          get_macrostr("signal", infop->si_status, NULL));
}
```

The output:

```
Process 9580 terminated:
        code =  CLD_EXITED
        exit value =  123
Process 9581 terminated:
        code =  CLD_DUMPED
        signal = SIGFPE
Process 9582 terminated:
        code =  CLD_KILLED
        signal = SIGHUP
```

With `waitid`, the problem of inadvertently getting a report on the wrong child can be averted with an algorithm like this:

1. Execute `waitid` with `P_ALL` and the `WNOWAIT` option.
2. If the returned process ID isn't one this part of the program is concerned with, go back to step 1.

3. If the process ID is of interest, reissue `waitid` for that process ID without the `WNOWAIT` option (or just use `waitpid`), to clear it out so it's no longer waitable.
4. If there are any other un-waited-for children to be waited for, go back to step 1.

The only problem with `waitid` is that it's not available on pre-SUS1 systems, which as of this writing include Linux, FreeBSD, and Darwin. So, the more awkward approaches described earlier in the `waitpid` discussion have to be used.

The behavior of `wait`, `waitpid`, and `waitid` is further affected by what the parent has done with the `SIGCHLD` signal, as explained in the next section.

5.9 Signals, Termination, and Waiting

Most of the details on signals are in Chapter 9, but it makes sense to talk about `SIGCHLD` here. The quick introduction to signals in Section 1.1.3 should be enough to understand this section; if not, skip it until after you've read Chapter 9.

As explained in the previous section, a parent can get the status of a child that changed state (terminated, stopped, or continued) with one of the `wait` variants. If a child isn't waited for at all, it stays waitable (a zombie) until the parent terminates.

I didn't mention it earlier, but a child that changes state normally sends a `SIGCHLD` signal to its parent. Unless the parent has made other arrangements, the default action is for the signal to be ignored, which is why it wasn't important to deal with it in the earlier examples.

The parent can catch the signal if it wants to be notified about changes in the state of a child. Unfortunately, the parent isn't told *which* child caused the signal.[12] If it uses one of the `wait` variants to get the process ID, there's a danger of getting the status of the wrong process, as explained in the previous section, because some other process could have changed state between the sending of the signal and the call to the `wait` variant. The safe thing to do is to use `waitid` (as explained in the previous section) or to use `waitpid` to check all of the possible children, one by one.

12. Unless the advanced features of the Realtime Signals Extension are used, but this extension isn't universally available.

Instead of waiting at all, a parent can prevent an unwaited-for child from becoming a zombie (becoming waitable) by calling `sigaction` with a flag of `SA_NOCLDWAIT` for the `SIGCHLD` signal. Then a child's status is simply tossed out when it terminates, and the parent therefore can't get it (and presumably doesn't care to). If the parent executes `wait` or `waitpid` anyway, regardless of its arguments (even `WNOHANG`), it blocks until *all* children terminate and then returns –1 with `errno` set to `ECHILD`. The parent can still be notified if it catches the `SIGCHLD` signal.

If the parent *explicitly* sets `SIGCHLD` signals to be ignored (say, calling `sigaction` with an action of `SIG_IGN`), as opposed to accepting the default action, it's the same as using the `SA_NOCLDWAIT` flag [X/Open]. But as not all systems support this, it's a good idea to set the `SA_NOCLDWAIT` flag also, which is more widely supported.

5.10 Implementing a Shell (Version 3)

With the `waitpid` version of the `display_status` function from Section 5.8, we can improve on `execute2` from Section 5.6:

```
static void execute3(int argc, char *argv[])
{
    pid_t pid;
    int status;

    switch (pid = fork()) {
    case -1: /* parent (error) */
        EC_FAIL
    case 0: /* child */
        execvp(argv[0], argv);
        EC_FAIL
    default: /* parent */
        ec_neg1( waitpid(pid, &status, 0) )
        display_status(pid, status);
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("execute3")
    if (pid == 0)
        _exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's some output; the command `fpe`, which does nothing more than try to divide by zero, is one written just for this example:

```
$ sh3
@ date
Tue Feb 25 12:49:52 MST 2003
Process 9954: Exit value 0
@ echo The shell is getting better!
The shell is getting better!
Process 9955: Exit value 0
@ fpe
Process 9956: Erroneous arithmetic operation - core dumped
@ EOT $
```

I've shown where I typed a Ctrl-d with the letters *EOT*.

5.11 Getting User and Group IDs

There are a bunch of systems calls to get the real and effective user and group IDs (concepts explained in Section 1.1.5):

getuid—get real user ID

```
#include <unistd.h>

uid_t getuid(void);
/* Returns user ID (no error return) */
```

geteuid—get effective user ID

```
#include <unistd.h>

uid_t geteuid(void);
/* Returns user ID (no error return) */
```

getgid—get real group ID

```
#include <unistd.h>

gid_t getgid(void);
/* Returns group ID (no error return) */
```

getegid—get effective group ID

```
#include <unistd.h>

gid_t getegid(void);
/* Returns group ID (no error return) */
```

A user or group ID is just some number. If you want a name, you have to call `getpwuid` or `getgrgid`, which were described in Section 3.5.2. Here's an example program that displays the real and effective user and group IDs:

```
int main(void)
{
    uid_t uid;
    gid_t gid;
    struct passwd *pwd;
    struct group *grp;

    uid = getuid();
    ec_null( pwd = getpwuid(uid) )
    printf("Real user = %ld (%s)\n", (long)uid, pwd->pw_name);

    uid = geteuid();
    ec_null( pwd = getpwuid(uid) )
    printf("Effective user = %ld (%s)\n", (long)uid, pwd->pw_name);

    gid = getgid();
    ec_null( grp = getgrgid(gid) )
    printf("Real group = %ld (%s)\n", (long)gid, grp->gr_name);

    gid = getegid();
    ec_null( grp = getgrgid(gid) )
    printf("Effective group = %ld (%s)\n", (long)gid, grp->gr_name);

    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

To make the output interesting, I changed the user and group owners of the program file (`uidgrp`) and then turned on the set-ID-on-execution bits for the user and group:

```
$ su
Password:
# chown adm:sys uidgrp
# chmod +s uidgrp
$ uidgrp
Real user = 100 (marc)
Effective user = 4 (adm)
Real group = 14 (sysadmin)
Effective group = 3 (sys)
$
```

5.12 Setting User and Group IDs

The rules for how the real and effective user and group IDs can be changed are complicated, and different depending on whether the process is running as superuser. Before explaining the rules, I need to note that in addition to its current real and effective user and group IDs, the kernel keeps for each process a record of the *original* effective user and group IDs that were set by the last `exec`. These are called the *saved set-user-ID* and *saved set-group-ID*.

Here now are the rules for setting user IDs; group IDs have similar rules:

1. Other than by an `exec` (which can change the saved ID), an ordinary (nonsuperuser) process can never explicitly change the real user ID or saved ID.
2. An ordinary process can change the effective ID to the real or saved ID.
3. A superuser process can change the real and effective user IDs to any user-ID value.
4. When a superuser process changes the real user ID, the saved ID changes to that value as well.

The superuser rules aren't that interesting—anything goes. The two rules for ordinary users essentially mean this: If an `exec` causes the real and effective user-IDs (or group IDs) to be different, the process can go back and forth between them.

Here's one scenario where going back and forth is useful: Imagine a utility `putfile` that transfers files between computers on a network. The utility needs access to a log file that only the administrative user (call it pfadm) can access. The program file is owned by pfadm, with the set-user-ID bit on. It starts running and can access the log file, since its effective user ID is pfadm. Then, to write the real user's files, it sets the effective user ID to the real user ID so that the real user's permissions will control the access. When it's done, it sets the effective user ID back to pfadm to access the log file again. (If the original effective user ID weren't saved, it wouldn't be able to go back.) Note that while it sounds like pfadm is special as far as this utility goes, it is just an ordinary user to the kernel—not a superuser.

Now for the calls themselves. The only useful ones for ordinary-user processes are `seteuid` and `setegid`.

seteuid—set effective user ID

```
#include <unistd.h>

int seteuid(
    uid_t uid              /* effective user ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setegid—set effective group ID

```
#include <unistd.h>

int setegid(
    gid_t gid              /* effective group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

According to the rules we laid out, for an ordinary-user process, the argument must be equal to the real user (or group) ID or to the saved ID. For a superuser, the arguments can be any user (or group) ID.

Superusers can use two additional calls:

setuid—set real, effective, and saved user ID

```
#include <unistd.h>

int setuid(
    uid_t uid              /* real, effective, and saved user ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setgid—set real, effective, and saved group ID

```
#include <unistd.h>

int setgid(
    gid_t gid              /* real, effective, and saved group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

For a superuser, the argument can be any user (or group) ID, and it sets the real, effective, and saved values. For an ordinary user, these two calls act the same as the "e" versions I just showed, which is very confusing, so they should be avoided.

Two more calls, `setreuid` and `setregid`, overlap the functionality of the four calls I already presented and add additional confusion, so they should also be avoided. (They were important in older systems, before `seteuid` and `setegid` were introduced.)

5.13 Getting Process IDs

A process can get its process ID and its parent's process ID with these calls:

getpid—get process ID

```
#include <unistd.h>

pid_t getpid(void);
/* Returns process ID (no error return) */
```

getppid—get parent process ID

```
#include <unistd.h>

pid_t getppid(void);
/* Returns parent process ID (no error return) */
```

I already used `getpid` in an example back in Section 2.4.3.

5.14 `chroot` System Call

chroot—change root directory

```
#include <unistd.h>

int chroot(
    const char *path          /* path name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

This system call, reserved for superuser processes, changes the root directory of the process, the directory whose path is /. It's nonstandard but implemented on essentially all UNIX-like systems (because it's very old).

There are two uses for `chroot` that come to mind:

- When a command with built-in path names (such as /etc/passwd) is to be run on other than the usual files. A new tree can be constructed wherever convenient; the root can be changed, and then the command can be executed. This is primarily useful in testing new commands before they are installed.
- For added security, such as when a web-server process begins processing an HTML file. It can set the root to the base path, which guarantees that no paths to other files can be accessed. Even the path ".." won't work because at the root it's interpreted as a reference right back to the root.

Once the root is changed, there's no way to get back with `chroot`. However, some systems provide a sister function `fchroot` that takes a file descriptor. (Analogous to the relationship between `fchdir` and `chdir`, as explained in Section 3.6.2.) You can capture a file descriptor open to the current root by opening it for reading, execute `chroot`, and then get back with `fchroot` using the saved file descriptor as an argument.

5.15 Getting and Setting the Priority

As I mentioned back in Section 1.1.6, each process has a *nice valu*e, which is a way for a process to influence the kernel's scheduling priority. Higher values mean that the process wants to be nice—that is, run at a lower priority. Lower values mean less nice—higher priority. To keep the nice values positive, they are offset from a number that varies with the system (strangely, it's referred to as NZERO in UNIX documentation); a typical value is 20. A process starts out with 20 and can be as nice as 39 or as not-nice as 0.

To be nice or not-nice, a process executes the `nice` system call:

nice—change nice value

```
#include <unistd.h>

int nice(
    int incr                /* increment */
);
/* Returns old nice value - NZERO or -1 on error (sets errno) */
```

The `nice` system call adds `incr` to the nice value. The resulting nice value must be between 0 and 39, inclusive; if an invalid value results, the nearest valid value is used. Only the superuser can lower the nice value, getting better-than-average service.

`nice` actually returns the new nice value minus 20, so the returned value is in the range –20 through 19 if NZERO is 20. However, the returned value is rarely of much use. This is just as well because a new nice value of 19 is indistinguishable from an error return (19 – 20 = –1). That this bug has remained unfixed, if not unnoticed, for so many years is indicative of how little most UNIX system programmers care about error returns from minor system calls like `nice`.[13]

13. Finally fixed in the latest version of [SUS2002]. You can set `errno` to zero before the call and then test it afterwards.

Most UNIX users are familiar with the nice *command,* which runs a program at a lower priority (or a higher one if run by a superuser). Here's our version:

```
#define USAGE "usage: aupnice [-num] command\n"

int main(int argc, char *argv[])
{
    int incr, cmdarg;
    char *cmdname, *cmdpath;

    if (argc < 2) {
        fprintf(stderr, USAGE);
        exit(EXIT_FAILURE);
    }
    if (argv[1][0] == '-') {
        incr = atoi(&argv[1][1]);
        cmdarg = 2;
    }
    else {
        incr = 10;
        cmdarg = 1;
    }
    if (cmdarg >= argc) {
        fprintf(stderr, USAGE);
        exit(EXIT_FAILURE);
    }
    (void)nice(incr);
    cmdname = strchr(argv[cmdarg], '/');
    if (cmdname == NULL)
        cmdname = argv[cmdarg];
    else
        cmdname++;
    cmdpath = argv[cmdarg];
    argv[cmdarg] = cmdname;
    execvp(cmdpath, &argv[cmdarg]);
    EC_FAIL

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Note how we reuse the incoming `argv` in the call to `execvp`. The variable `cmdarg` holds the subscript of the command path (either 1 or 2, depending on whether an increment was specified). That part of `argv` starting with `cmdarg` is the part to be passed on. You might want to contrast this with the more extensive

manipulation needed for `execvp2` in Section 5.3. There we had to *insert* an argument, which required us to recopy the entire array.

The first string in the vector is supposed to be just the command name, not the whole path, so we had to strip off all but the last component of the path, in case the command was executed with a pathname containing slashes. The first argument to `execvp`, though, has to be whatever was typed on the `aupnice` command line.

Note also that we used `execvp` without a `fork`. Having changed the priority, there's no need to keep a parent alive just to wait around.

Here's `aupnice` in action on Solaris. For the example command, it uses `ps` reporting on itself. Note that the internal priority (whatever that is) changed from 58 to 28 when the nice value was increased by 10 (`aupnice`'s default).

```
$ ps -ac
   PID  CLS PRI TTY      TIME CMD
 10460   TS  58 pts/2    0:00 ps
$ aupnice ps -ac
   PID  CLS PRI TTY      TIME CMD
 10461   TS  28 pts/2    0:00 ps
```

There are some newer and fancier calls called `getpriority` and `setpriority` that you can use to manipulate the nice value; see [SUS2002] or your system's documentation for details.

5.16 Process Limits

The kernel enforces various limits on process resources, such as the maximum file size and the maximum stack size. Usually, when a limit is reached, whatever function (e.g., `write`, `malloc`) was the cause returns an error, or a signal is generated, such as `SIGSEGV` in the case of stack overflow (Chapter 9).[14] There are seven standard resources, listed below, whose limits can be set or gotten, and implementations may define others.

14. If `SIGSEGV` is caught, how can the signal handler execute when there's no stack? You can set up an alternate stack, as explained in Section 9.3. If you haven't done so, the signal is set back to its default behavior, which is to terminate the process.

For each resource, there's a *maximum* (or *hard*) limit, and a *current* (or *soft*) limit. The current limit is what's effective, and an ordinary process is allowed to set it to any reasonable number up to the maximum. An ordinary process can also irreversibly lower (but not raise) the maximum to whatever the current limit is. A superuser process can set either limit to whatever the kernel will support.

Here are the system calls to get and set the limits:

getrlimit—get resource limits

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

int getrlimit(
    int resource,              /* resource */
    struct rlimit *rlp         /* returned limits */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setrlimit—set resource limits

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

int setrlimit(
    int resource,              /* resource */
    const struct rlimit *rlp   /* limits to set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct rlimit—structure for getrlimit and setrlimit

```
struct rlimit {
    rlim_t rlim_cur;           /* current (soft) limit */
    rlim_t rlim_max;           /* maximum (hard) limit */
};
```

Each call is for a single resource; you get its limits with `getrlimit`, and set them, under the rules given previously, with `setrlimit`. The standard resources are:

`RLIMIT_CORE` Maximum size of a core (memory-dump) file in bytes; zero means no file is written at all. There's no error reported if it's exceeded—writing just stops at that size, probably leaving a useless file.

`RLIMIT_CPU` Maximum CPU time in seconds. Exceeding it generates a `SIGXCPU`, which by default terminates the process. The standard doesn't define what happens if you catch, ignore, or block this signal—the process may be terminated, as there's no way to continue without accumulating more CPU time.

RLIMIT_DATA	Maximum size of data segment, in bytes. Exceeding it causes the memory-allocation function (e.g., malloc) to fail.
RLIMIT_FSIZE	Maximum file size in bytes. Exceeding it generates a SIGXFSZ; if that signal is caught, ignored, or blocked, the offending function fails.
RLIMIT_NOFILE	Maximum number of file descriptors that a process may use.[15] Exceeding it causes the offending function to fail.
RLIMIT_STACK	Maximum stack size in bytes. Generates a SIGSEGV if it's exceeded.
RLIMIT_AS	Maximum total memory size, including the data segment, the stack, and anything mapped in with mmap (Section 7.14). Exceeding it causes the offending function to fail or, if it's the stack, the effect described for RLIMIT_STACK.

Process limits are preserved across an exec and inherited by the child after a fork, so if they're set once at login time, they'll affect all processes descended from the login shell. But, like cd, the command to set them has to be built into the shell—executing it in a child of the shell would be ineffective. Most shells have a ulimit command that sets the file-size limit, and other shells have a more general command called perhaps limit, limits, or plimit.[16]

There are some special macros for the rlim_cur and rlim_max members of an rlimit structure, in addition to actual numbers: RLIM_SAVED_CUR, RLIM_SAVED_MAX, and RLIM_INFINITY.

When you call getrlimit, the actual numbers are returned if they fit into an rlim_t, which is an unsigned type whose width isn't specified by the standard. If a number won't fit, the member is set to RLIM_SAVED_MAX if it's equal to the maximum. Otherwise it's set to RLIM_SAVED_CUR, which means "the limit is set to what the limit is set to." If a member is set to RLIM_INFINITY, it means there is no limit.

15. Technically, it's one greater than the maximum file-descriptor number. All the potential file descriptors count toward the limit even if they're not opened.

16. Don't confuse these *process* limits with *user* limits, such as user's disk quota. User limits are queried and set by nonstandard commands such as quota, implemented in terms of a nonstandard system call such as quotactl or ioctl.

When you call `setrlimit`, if a member is `RLIM_SAVED_CUR`, it's set to the current limit; if it's `RLIM_SAVED_MAX`, it's set to the maximum limit; if it's `RLIM_INFINITY`, no limit will be enforced. Otherwise, the limit is set to whatever number you've used. However, you can use `RLIM_SAVED_CUR` or `RLIM_SAVED_MAX` only if a call to `getrlimit` returned those values; otherwise, you have to use actual numbers. To say it differently: You can use those macros only when you're forced to because `rlim_t` won't hold the number.

The `RLIM_SAVED_CUR` and `RLIM_SAVED_MAX` rigmarole is really just this: You can manipulate the limits to a degree even if you can't get at their actual values.

If an implementation will never have a limit that won't fit in an `rlim_t`, the `RLIM_SAVED_CUR` or `RLIM_SAVED_MAX` macros will never be used, so they're allowed to have the same value as `RLIM_INFINITY`. We'll see the impact of this in the following example, which displays the limits, sets the file limit to a ridiculously low number, and then exceeds it, causing the program to be terminated by a `SIGXFSZ`:

```
int main(void)
{
    struct rlimit r;
    int fd;
    char buf[500] = { 0 };

    if (sizeof(rlim_t) > sizeof(long long))
        printf("Warning: rlim_t > long long; results may be wrong\n");
    ec_false( showlimit(RLIMIT_CORE, "RLIMIT_CORE") )
    ec_false( showlimit(RLIMIT_CPU, "RLIMIT_CPU") )
    ec_false( showlimit(RLIMIT_DATA, "RLIMIT_DATA") )
    ec_false( showlimit(RLIMIT_FSIZE, "RLIMIT_FSIZE") )
    ec_false( showlimit(RLIMIT_NOFILE, "RLIMIT_NOFILE") )
    ec_false( showlimit(RLIMIT_STACK, "RLIMIT_STACK") )
#ifndef FREEBSD
    ec_false( showlimit(RLIMIT_AS, "RLIMIT_AS") )
#endif
    ec_neg1( getrlimit(RLIMIT_FSIZE, &r) )
    r.rlim_cur = 500;
    ec_neg1( setrlimit(RLIMIT_FSIZE, &r) )
    ec_neg1( fd = open("tmp", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
    ec_neg1( write(fd, buf, sizeof(buf)) )
    ec_neg1( write(fd, buf, sizeof(buf)) )
    printf("Wrote two buffers! (?)\n");
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
```

```
EC_CLEANUP_END
}

static bool showlimit(int resource, const char *name)
{
    struct rlimit r;

    ec_neg1( getrlimit(resource, &r) )
    printf("%s: ", name);
    printf("rlim_cur = ");
    showvalue(r.rlim_cur);
    printf("; rlim_max = ");
    showvalue(r.rlim_max);
    printf("\n");
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void showvalue(rlim_t lim)
{
    /*
        All macros may equal RLIM_INFINITY; that test
        must be first; can't use switch statement.
    */
    if (lim == RLIM_INFINITY)
        printf("RLIM_INFINITY");
#ifndef BSD_DERIVED
    else if (lim == RLIM_SAVED_CUR)
        printf("RLIM_SAVED_CUR");
    else if (lim == RLIM_SAVED_MAX)
        printf("RLIM_SAVED_MAX");
#endif
    else
        printf("%llu", (unsigned long long)lim);
}
```

FreeBSD and Darwin don't implement the whole thing, as you can see from the code.[17]

Here's what I got on our four test systems, starting with Solaris:

```
RLIMIT_CORE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
```

17. Actually, these calls originated with 4.2BSD, but the BSD-derived systems haven't picked up some of the more recent POSIX extensions.

```
RLIMIT_DATA: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 256; rlim_max = 1024
RLIMIT_STACK: rlim_cur = 8683520; rlim_max = 133464064
RLIMIT_AS: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
File Size Limit Exceeded - core dumped
```

Linux:

```
RLIMIT_CORE: rlim_cur = 0; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 1024; rlim_max = 1024
RLIMIT_STACK: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_AS: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
File size limit exceeded
```

FreeBSD:

```
RLIMIT_CORE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = 536870912; rlim_max = 536870912
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 957; rlim_max = 957
RLIMIT_STACK: rlim_cur = 67108864; rlim_max = 67108864
Filesize limit exceeded
```

Darwin:

```
RLIMIT_CORE: rlim_cur = 0; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = 6291456; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 256; rlim_max = RLIM_INFINITY
RLIMIT_STACK: rlim_cur = 524288; rlim_max = 67108864
Filesize limit exceeded
```

There's a much older system call that's less functional and has problems with its return value:

ulimit—get and set process limits

```
#include <ulimit.h>

long ulimit(
     int cmd,               /* command */
     ...                    /* optional argument */
);
/* Returns limit or -1 with errno changed on error (sets errno) */
```

The cmd argument is one of:

UL_GETFSIZE Get the current file-size limit; equivalent to getrlimit with RLIMIT_FSIZE.

UL_SETFSIZE Set the file-size limit to the second long argument; it may be increased only by a superuser process. Equivalent to setrlimit with RLIMIT_FSIZE where rlim_max is adjusted and rlim_cur is adjusted to match.

The sizes set or gotten are in units of 512 bytes. Because the sizes can get big, the returned value might be negative, and even –1 is possible. So, the way to check for an error is to set errno to zero before the call. If it returns –1, it's an error only if errno changed.

There are some more peculiarities with ulimit, but there's no point going into them. It's basically a mess, and you should use getrlimit and setrlimit instead.

The getrusage system call gets information about resource usage by a process or by its terminated and waited-for children:

getrusage—get resource usage

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

int getrusage(
     int who,                    /* RUSAGE_SELF or RUSAGE_CHILDREN */
     struct rusage *r_usage      /* returned usage information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct rusage—structure for getrusage

```
struct rusage {
     struct timeval ru_utime;   /* user time used */
     struct timeval ru_stime;   /* system time used */
     /* following members are nonstandard */
     long ru_maxrss;            /* maximum resident set size */
     long ru_ixrss;             /* integral shared memory size */
     long ru_idrss;             /* integral unshared data size */
     long ru_isrss;             /* integral unshared stack size */
     long ru_minflt;            /* page reclaims */
     long ru_majflt;            /* page faults */
     long ru_nswap;             /* swaps */
     long ru_inblock;           /* block input operations */
     long ru_oublock;           /* block output operations */
     long ru_msgsnd;            /* messages sent */
     long ru_msgrcv;            /* messages received */
     long ru_nsignals;          /* signals received */
     long ru_nvcsw;             /* voluntary context switches */
     long ru_nivcsw;            /* involuntary context switches */
};
```

[SUS2002] only specifies the first two members. Newer BSD systems, including FreeBSD, support them all. Solaris 8 (on Intel, anyway) supports only the standard members; Linux 2.4 supports the standard members plus `ru_minflt`, `ru_majflt`, and `ru_nswap`. For details about what the members mean, check your system's `man` page for `getrusage`; for the whole story, run it on a BSD-based system if you can.

The two times returned are similar to what `times` returns (Section 1.7.2), but to a higher resolution, since a `timeval` (Section 1.7.1) is in microseconds.

5.17 Introduction to Threads

This section gives a very brief introduction to threads, just enough to show some interesting example programs. My goals are to explain what they are, help you decide if your application design can benefit from them, and caution you about the hazards of using them if you're not careful.

For the whole truth about threads (there are about a hundred system calls to manage them), you'll have to pick up a book on the subject, such as [Nor1997] or [But1997].

5.17.1 Thread Creation

In our examples up to now, the processes we created (with `fork`) had only one flow of control, or *thread,* in them. Execution proceeded from instruction to instruction, in a single sequence, and the stack, global data, and system resources got modified by the various instructions, some of which may have executed system calls.

With the POSIX Threads feature of UNIX, a process can have multiple threads, each of which has its own flow of control (its own instruction counter and CPU clock) and its own stack. Essentially everything else about the process, including global data and resources such as open files or the current directory, is shared.[18] The following code shows what I mean.

18. A thread can also arrange to have some thread-specific data; see [SUS2002] or one of the referenced books for details.

```
static long x = 0;

static void *thread_func(void *arg)
{
    while (true) {
        printf("Thread 2 says %ld\n", ++x);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid;

    ec_rv( pthread_create(&tid, NULL, thread_func, NULL) )
    while (x < 10) {
        printf("Thread 1 says %ld\n", ++x);
        sleep(2);
    }
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The sequence in which the threads print is unpredictable; this is the output we happened to get:

```
Thread 2 says 1
Thread 1 says 2
Thread 2 says 3
Thread 1 says 4
Thread 2 says 5
Thread 2 says 6
Thread 1 says 7
Thread 2 says 8
Thread 2 says 9
Thread 1 says 10
Thread 2 says 11
Thread 2 says 12
```

So you can see that both threads—the one in `main` and the one in `thread_func`—accessed the same global variable, `x`. There are some problems with this that I'll address in Section 5.17.3.

The initial thread—the one containing `main`—kicked off the second thread with the `pthread_create` system call:

pthread_create—create thread

```
#include <pthread.h>

int pthread_create(
    pthread_t *thread_id,          /* new thread's ID */
    const pthread_attr_t *attr,    /* attributes (or NULL) */
    void *(*start_fcn)(void *),    /* starting function */
    void *arg                      /* arg to starting function */
);
/* Returns 0 on success, error number on error */
```

The new thread starts with a call to the start function specified in the `pthread_create` call, which must have this prototype:

pthread starting function

```
void *start_fcn(
    void *arg
);
/* Returns exit status */
```

Whatever you pass as the fourth argument to `pthread_create` is passed directly to the starting function. It's a `void` pointer for generality, and often it will indeed be a pointer to some data. Since the threads share the same address space, the pointer is valid for both threads. If you want, you can also pass in integer data, but you have to cast it to a `void` pointer. And, to be safe, you should check (with an `assert`, say) that the integer type you're using fits in a `void` pointer, with a line like this that you'll see in some of the examples that follow:

```
assert(sizeof(long) <= sizeof(void *));
```

The attributes that affect the new thread are not just some flags but rather an object of type `pthread_attr_t` that you have to set with calls like `pthread_attr_setscope` and `pthread_attr_setstacksize`, which I won't go into here. (But perhaps you're beginning to appreciate why there are a hundred thread-related calls.) In our examples we'll use just default attributes, so our second argument to `pthread_create` will be `NULL`.

The "pthread" functions uniformly return 0 on success and an error code on failure, rather than setting `errno`. We have a special "ec" macro for this, `ec_rv`, which takes the error code, if any, and treats it as if it were an `errno` value, which it is.[19]

19. The various functions in the error-checking package were modified to be thread-safe, but the code for that isn't shown in this book. You'll see it on the Web site, however.

5.17.2 Waiting for a Thread to Terminate

A thread can wait for another thread to terminate and get its exit status with `pthread_join` (analogous to `wait` and its variants).

pthread_join—wait for thread to terminate

```
#include <pthread.h>

int pthread_join(
    pthread_t thread_id,   /* ID of thread to join */
    void **status_ptr      /* returned exit status (if not NULL arg) */
);
/* Returns 0 on success, error number on error */
```

Here's my little example modified so that Thread 1 passes a limit to Thread 2, which then reports the value of `x` back to Thread 1:

```
static long x = 0;

static void *thread_func(void *arg)
{
    while (x < (long)arg) {
        printf("Thread 2 says %ld\n", ++x);
        sleep(1);
    }
    return (void *)x;
}

int main(void)
{
    pthread_t tid;
    void *status;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (x < 10) {
        printf("Thread 1 says %ld\n", ++x);
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The output:

```
Thread 1 says 1
Thread 2 says 2
Thread 2 says 3
Thread 1 says 4
Thread 2 says 5
Thread 2 says 6
Thread 1 says 7
Thread 1 says 8
Thread 1 says 9
Thread 1 says 10
Thread 2's exit status is 7
```

5.17.3 Thread Synchronization (Mutexes)

As we're going to see in the next few chapters, the challenge on UNIX of working with multiple processes (perhaps across a network) is allowing them to *share* data. With threads it's the complete opposite—the challenge is to keep their natural sharing of data *separated*. In fact, the two thread examples I've shown are defective in that both threads might simultaneously access the same data, the variable `x`. While it might seem that a simple increment operator is an atomic operation, there's no guarantee that it is. It's actually possible for Thread 1 to update half of a 32-bit `x` while Thread 2 reads the full 32 bits, getting a mishmash instead of a valid integer.[20] With more complicated shared data structures—a much more realistic situation—the problem is much worse. We want the access to shared data to be atomic. Two examples of what this means are:

- If updating a data structure leaves it in a temporarily inconsistent state, no thread other than the one updating sees it in that state, and
- If a thread has to read data, compute results, and write them back, no other thread will modify the data until the entire sequence is complete. Otherwise, the other thread's modification would be lost.

These requirements are provided for by some simple system calls that implement mutual-exclusion objects, called *mutexes* for short. Threads use them to protect critical sections where another thread may otherwise see inconsistent data or interfere with updating. The principal mutex system calls are `pthread_mutex_lock` and `pthread_mutex_unlock`:

20. And that's only one thing that can go wrong. Another is that compiler optimization might leave the integer in a register. You really can't ever let threads simultaneously access data without protection.

pthread_mutex_lock—lock mutex

```
#include <pthread.h>

int pthread_mutex_lock(
    pthread_mutex_t *mutex    /* mutex to lock */
);
/* Returns 0 on success, error number on error */
```

pthread_mutex_unlock—unlock mutex

```
#include <pthread.h>

int pthread_mutex_unlock(
    pthread_mutex_t *mutex    /* mutex to unlock */
);
/* Returns 0 on success, error number on error */
```

Mutual exclusion works like this: If a mutex is already locked, `pthread_mutex_lock` blocks until it is unlocked.

The simplest way to get a suitable mutex variable is just to declare it with an initializer, like this:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

It can be local to the file (`static`) or shared between files (`extern`). You can also have mutexes on the stack (automatic variables) or allocated dynamically, but then you need to initialize them with a call to `pthread_mutex_init`, which I won't detail here.[21]

Now we can modify our example to adequately protect the shared data:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static long x = 0;

static void *thread_func(void *arg)
{
    bool done;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        done = x >= (long)arg;
        ec_rv( pthread_mutex_unlock(&mtx) )
```

21. You're not supposed to initialize an automatic mutex variable with the initializer `PTHREAD_MUTEX_INITIALIZER`, even though the compiler will let you, because on some systems that might invoke a function that's not thread safe. In C++, you can get into similar trouble if you use `PTHREAD_MUTEX_INITIALIZER` to initialize a static internal mutex, since C++ might delay the initialization until the function is called.

```
        if (done)
            break;
        ec_rv( pthread_mutex_lock(&mtx) )
        printf("Thread 2 says %ld\n", ++x);
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(1);
    }
    return (void *)x;

EC_CLEANUP_BGN
    EC_FLUSH("thread_func")
    return NULL;
EC_CLEANUP_END
}

int main(void)
{
    pthread_t tid;
    void *status;
    bool done;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        done = x >= 10;
        ec_rv( pthread_mutex_unlock(&mtx) )
        if (done)
            break;
        ec_rv( pthread_mutex_lock(&mtx) )
        printf("Thread 1 says %ld\n", ++x);
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

Note that we had to pull the conditionals out of the `while` expressions so that we could surround them with the locking and unlocking calls. We certainly don't want to protect the whole loop, for that would destroy the concurrency. It requires some care to protect enough to be safe but not so much to hurt performance. Unfortunately, while you may be able to detect a performance problem with the

right kind of testing, it's much harder to test whether you've prevented all race conditions.

We still have a defect: Suppose in `pthread_func` the call to `pthread_mutex_unlock` fails, causing the thread to exit. This might leave the mutex locked, causing the initial thread to block forever in one of its calls to `pthread_mutex_lock`. A potential solution is to make sure the mutex is unlocked in `pthread_func`'s cleanup code, with another call to `pthread_mutex_unlock`, like this:

```
EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return NULL;
EC_CLEANUP_END
```

We're assuming that if the first call to `pthread_mutex_unlock` failed, the second one might succeed, which seems like a stretch. Since there is really no way either `pthread_mutex_lock` or `pthread_mutex_unlock` can fail as long as the mutex has a valid value, probably the easiest, safest, and clearest way to proceed is to check the error return from these functions during debugging, but not in the production program. Or, you might decide that for your application it's better to leave the error checks in, as long as you get a report of the error, so you don't waste time trying to find out why the application stalled. This is another example of why threads can be very tricky to use correctly. You may find that you spend 5% of your time implementing the threads and 95% of your time ensuring that you've done it right.

The proper way to proceed is to encapsulate the access to shared data with a collection of functions (or with a class, if you're using an object-oriented language like C++) with the mutexes in the access functions. Don't spread the locking and unlocking calls willy-nilly throughout your program, as I did.

So I'll show the example rewritten yet again, this time with all access to `x` going through a single function, `get_and_incr_x`, which does all the locking and unlocking. Both `x` and the mutex have been moved inside the function. Notice how much more readable this version is than the previous one. More readable almost always implies more reliable!

```
static long get_and_incr_x(long incr)
{
    static long x = 0;
    static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
    long rtn;
```

```
    ec_rv( pthread_mutex_lock(&mtx) )
    rtn = x += incr;
    ec_rv( pthread_mutex_unlock(&mtx) )
    return rtn;

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void *thread_func(void *arg)
{
    while (get_and_incr_x(0) < (long)arg) {
        printf("Thread 2 says %ld\n", get_and_incr_x(1));
        sleep(1);
    }
    return (void *)get_and_incr_x(0);
}

int main(void)
{
    pthread_t tid;
    void *status;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (get_and_incr_x(0) < 10) {
        printf("Thread 1 says %ld\n", get_and_incr_x(1));
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

This time I decided to make locking or unlocking errors fatal, which is another possible choice. My point certainly isn't that this is what you should be doing in all cases. Rather, the point is that there are many ways to handle these kinds of errors, and what's best depends on the needs of the specific application.

It's important to understand why these lines in function `get_and_incr_x` work:

```
ec_rv( pthread_mutex_lock(&mtx) )
rtn = x += incr;
ec_rv( pthread_mutex_unlock(&mtx) )
```

It's the access to `x`, which, as a global variable, is shared between the threads that we need to protect. The variables `rtn` and `incr` are on a stack unique to each thread (that is, each thread has its own copies) and need no protection.

I have one more observation about this example which I'll defer to Exercise 5.11.

There are three other kinds of thread-synchronization objects that you can read about in [SUS2002] or one of the books referenced at the start of Section 5.17:

- *Read-write locks* are like mutexes, but they distinguish between locking against reading the data and locking against writing, to gain additional concurrency. See Section 7.11.4 for more about read-write locks in general.
- *Spin locks* are also like mutexes, but they're faster and intended for short durations. They're usually implemented by testing the lock in a CPU loop instead by blocking the thread.
- *Barriers* are synchronization points at which one or more threads wait, ensuring that they have all completed some task before any one is allowed to continue.

5.17.4 Condition Variables

Suppose Thread A is doing some work (reading data from a network connection, say) and adding items to a queue, and Thread B is taking items from the queue and doing some additional work on them (updating a database, say). Using a mutex *M* to control access to the queue, you could organize the threads like this:

Thread A	*Thread B*
1. Read data [B]	1. Lock *M* [b]
2. Lock *M* [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock *M*
4. Unlock *M*	4. Goto step 1
5. Goto step 1	

(The notation [B] means an indeterminate block, and [b] means a short-term block.)

This works OK, but Thread B wastes a lot of CPU time when the queue is empty, as it keeps checking while it loops and loops. You could slow it down a bit like this:

Thread A	*Thread B*
1. Read data [B]	1. Lock *M*
2. Lock *M* [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock *M* [b]
4. Unlock *M*	4. Sleep for 1 sec. [b]
5. Goto step 1	5. Goto step 1

But this isn't much of an improvement: Thread B is now less responsive because it may be sleeping when there's work to do, and it still wastes CPU time because the queue may be empty when it wakes up. What we want to do is have Thread A signal Thread B when it puts an item on the queue and have Thread B block until it gets the signal. Let's try it with another mutex *Q* to represent the concept "queue is nonempty":

Thread A	*Thread B*
1. Read data [B]	1. Lock *M* [b]
2. Lock *M* [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock *M*
4. Unlock *M*	4. Lock *Q* [B]
5. Unlock *Q*	5. Goto step 1
6. Goto step 1	

Thread B's attempt to lock *Q* (step 4) is an indeterminate block because *Q* becomes unlocked only when Thread A returns from its read (step 1).

Now the problem is that Thread A may try to unlock *Q* when Thread B hasn't got it locked, which would be a lost signal—the attempt to unlock isn't somehow remembered for the next attempt to lock (i.e., mutexes are not counting semaphores). So if this happens, Thread B will be held up waiting for the lock on *Q*, and the lock won't be released until the next time data is read, if ever. Meanwhile, the item on the queue goes unprocessed by Thread B.

A more concrete problem with *Q* is that only the thread that locks a mutex can unlock it. So, A's step 5 would result in an error.

We could try to replace *Q* with a counting semaphore (Section 7.8), but a better choice is to use another kind of signaling mechanism that POSIX Threads provides: *condition variables*. Here's how they're used in this example:

Thread A	*Thread B*
1. Read data [B]	1. Lock *M* [b]
2. Lock *M* [b]	2. while (queue is empty) {
3. Put item on queue	cond_wait(*C*, *M*) [B]
4. cond_signal(*C*)	}
5. Unlock *M*	3. Remove item; update database
6. Goto step 1	4. Unlock *M*
	5. Goto step 1

The second mutex, *Q*, is gone, and now there's a condition variable, *C*. In Thread B, cond_wait waits until condition *C* is signaled, which it is in Thread A's step 4. But cond_wait has a special interaction with mutex *M* (its second argument): The mutex must be locked when cond_wait is called. *It is unlocked during the waiting, and then relocked automatically when cond_wait returns.* That way there's no possibility of a missed signal or of deadlock, both of which were defects in our earlier attempts. And, all the code in Thread B's steps 2 and 3 is executed with *M* locked, as it should be.

Why is cond_wait in a loop that tests whether the queue is empty, when condition *C* gets signaled (by Thread A) only when the queue is nonempty? It's because cond_wait, like any UNIX blocking system call, is subject to being interrupted (e.g., by the arrival of a traditional UNIX signal, such as `SIGINT`), in which case there might be a return with the queue still empty. A loop that tests the predicate (empty queue, in this case) ensures that we'll go right back into cond_wait on such a spurious return. Because spurious returns are infrequent, the wasted CPU time is inconsequential.

Another reason for checking the queue before calling cond_wait is that cond_signal might have been called well before Thread B calls cond_wait, and a con_signal with no thread waiting is discarded (*not* held pending). So, it's important not to call cond_wait unless waiting is necessary, or the wait may be forever, even though the queue is nonempty.

Thus, to signal and wait for a condition, you need three things: a *condition variable*, a *mutex*, and a *predicate*. The first two are special data types, but the

predicate is just some ordinary code that makes sense for your program—it's the concrete details of what the condition variable represents abstractly.

Here are the synopses for the actual `pthread_cond_signal` and `pthread_cond_wait` system calls:

pthread_cond_signal—signal condition

```
#include <pthread.h>

int pthread_cond_signal(
    pthread_cond_t *cond      /* condition variable */
);
/* Returns 0 on success, error number on error */
```

pthread_cond_wait—wait for condition

```
#include <pthread.h>

int pthread_cond_wait(
    pthread_cond_t *cond,     /* condition variable */
    pthread_mutex_t *mutex    /* mutex */
);
/* Returns 0 on success, error number on error */
```

As with a mutex, you can declare and initialize a condition variable statically, like this:

```
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

A condition variable on the stack (automatic) or allocated dynamically needs to be initialized with a call to `pthread_cond_init`, which I won't go into in this book.

Here's an example program that has the initial thread putting nodes onto a queue while another thread takes them off and displays their contents. The initial thread signals a condition when a node is queued, and the other thread waits on that condition:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

struct node {
    int n_number;
    struct node *n_next;
} *head = NULL;
```

```
static void *thread_func(void *arg)
{
    struct node *p;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return (void *)false;
EC_CLEANUP_END
}

int main(void)
{
    pthread_t tid;
    int i;
    struct node *p;

    ec_rv( pthread_create(&tid, NULL, thread_func, NULL) )
    for (i = 0; i < 10; i++) {
        ec_null( p = malloc(sizeof(struct node)) )
        p->n_number = i;
        ec_rv( pthread_mutex_lock(&mtx) )
        p->n_next = head;
        head = p;
        ec_rv( pthread_cond_signal(&cond) )
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(1);
    }
    ec_rv( pthread_join(tid, NULL) )
    printf("All done -- exiting\n");
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The guts of the initial thread that queues a new node follows the pattern shown earlier—it calls `pthread_cond_signal` with the mutex locked:[22]

```
ec_rv( pthread_mutex_lock(&mtx) )
p->n_next = head;
head = p;
ec_rv( pthread_cond_signal(&cond) )
ec_rv( pthread_mutex_unlock(&mtx) )
```

And the thread that removes a node also follows the pattern, calling `pthread_cond_wait` with the mutex locked:

```
ec_rv( pthread_mutex_lock(&mtx) )
while (head == NULL)
    ec_rv( pthread_cond_wait(&cond, &mtx) )
p = head;
head = head->n_next;
printf("Got %d from front of queue\n", p->n_number);
free(p);
ec_rv( pthread_mutex_unlock(&mtx) )
```

These two threads work successfully only because `pthread_cond_wait` unlocks the mutex while it's waiting and then relocks it before returning. So, the following is guaranteed to be true:

- Both critical sections that deal with the queue are executed under the protection of the mutex `mtx`.
- When the statement

  ```
  p = head;
  ```

  is executed in the second thread, `head` is non-`NULL`.

Here's the output we got:

```
Got 0 from front of queue
Got 1 from front of queue
Got 2 from front of queue
Got 3 from front of queue
Got 4 from front of queue
Got 5 from front of queue
Got 6 from front of queue
Got 7 from front of queue
Got 8 from front of queue
Got 9 from front of queue
```

What happened to the "All done – exiting" message? We never got it. Actually, the program hung after printing the "Got 9" line, and we typed Ctrl-C to stop it.

22. It's also OK to call `pthread_cond_signal` with the mutex unlocked, and in fact that might increase throughput a bit.

Looking at the program, we can see why: The second thread stays in its loop forever, looking for a node on the queue that will never arrive, and the initial thread never returns from its call to `pthread_join`.

An obvious fix is to put some sort of "end-of-file" node on the queue to tell the second thread to exit. Or, the initial thread can just cancel the second thread when there's going to be no more work. We'll do it that way so we have an excuse to talk about how to cancel a thread.

5.17.5 Canceling a Thread

One thread can cancel another thread with `pthread_cancel`:

pthread_cancel—cancel thread

```
#include <pthread.h>

int pthread_cancel(
    pthread_t thread_id       /* ID of thread to cancel */
);
/* Returns 0 on success, error number on error */
```

Normally, the thread to be cancelled doesn't stop right away, but only at a *cancellation point,* which is when the thread calls one of the 200 or so system calls or standard functions that can block, such as `read`, `waitpid`, or `pthread_cond_wait`.[23] If a thread calls a function defined elsewhere in the program, or in a library, there's a pretty good chance of it calling one of those 200+ system calls or functions; therefore, you should consider any function call as a potential, but not guaranteed, cancellation point unless it is specifically documented otherwise and you trust the documentation.

The significance of cancellation points is that you can safely execute ordinary code without worrying about it being cancelled. For example, you can modify a linked list (possibly protected by a mutex), knowing that the code sequence will be allowed to complete.

You'll be relieved to know that none of the "pthread" mutex calls are cancellation points, nor are `free`, `calloc`, `malloc`, or `realloc`. If the mutex calls were cancellation points, it would be very cumbersome to use mutexes at all, as you would have to add code to handle cancellation every time you called `pthread_mutex_lock`.

23. The SUS specifies 65 or so that are *always* cancellation points and another 150 or so that *may* be cancellation points.

If a thread has no cancellation points at all, or none that you can depend on, yet stays alive long enough so that allowing cancellation is an issue, you can put in one or more calls to `pthread_testcancel` at safe places to explicitly provide cancellation points. This call doesn't do anything if there's no pending cancellation.

pthread_testcancel—test for cancellation

```
#include <pthread.h>

void pthread_testcancel(void);
```

I said that *normally* a thread is cancelled only at a cancellation point. This is true when its cancellation type is `PTHREAD_CANCEL_DEFERRED`, which is the default. If you set the type to `PTHREAD_CANCEL_ASYNCHRONOUS` with the `pthread_setcanceltype` function (see [SUS2002] for details), it really will be cancelled immediately, which is a scary thought. Presumably, anyone who would do that knows what he or she is doing.

Now we can modify the `main` function from the previous example to cancel the other thread when no more nodes will be queued:

```
for (i = 0; i < 10; i++) {
    ec_null( p = malloc(sizeof(struct node)) )
    p->n_number = i;
    ec_rv( pthread_mutex_lock(&mtx) )
    p->n_next = head;
    head = p;
    ec_rv( pthread_cond_signal(&cond) )
    ec_rv( pthread_mutex_unlock(&mtx) )
    sleep(1);
}
ec_rv( pthread_cancel(tid) )
ec_rv( pthread_join(tid, NULL) )
printf("All done -- exiting\n");
return EXIT_SUCCESS;
```

With this improvement the program terminates cleanly:

```
Got 0 from front of queue
Got 1 from front of queue
Got 2 from front of queue
Got 3 from front of queue
Got 4 from front of queue
Got 5 from front of queue
```

```
Got 6 from front of queue
Got 7 from front of queue
Got 8 from front of queue
Got 9 from front of queue
All done -- exiting
```

We're not quite done, though. We need to look carefully at the code for the thread that got cancelled to make sure that cancellation won't leave the queue in an inconsistent state. Here it is again:

```
static void *thread_func(void *arg)
{
    struct node *p;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return (void *)false;
EC_CLEANUP_END
}
```

Indeed, there are two problems:

- `pthread_cond_wait` is a cancellation point. If the thread is cancelled there, the queue is OK, as nothing yet has been done to it, but the mutex is relocked, which always is the case when leaving `pthread_cond_wait`. We don't want the thread to terminate with the mutex locked, as it can then never be unlocked.
- `printf` may be a cancellation point. If the thread terminates there, the node just removed won't be freed, resulting in a memory leak.

True, this program exits after cancelling the thread, but that may not always be the case so it makes sense to fix the bugs. It's important to protect against what *could* happen, not only against what usually happens.

A solution for the second problem is to store the number to be printed in a local variable, free the node, and then call `printf`.

But the first problem has no obvious solution. We have to install a *cancellation cleanup handler,* which is a function that's called just before cancellation to clean thing up. We'll fix the second problem in the cancellation cleanup handler as well.

A cancellation cleanup handler always has this prototype (the name doesn't matter):

```
void cleanup_handler(void *arg);
```

A thread installs a cleanup handler with `pthread_cleanup_push` and uninstalls it with `pthread_cleanup_pop`:

pthread_cleanup_push—install cleanup handler

```
#include <pthread.h>

void pthread_cleanup_push(
    void (*handler)(void*),    /* pointer to cleanup-handler function */
    void *arg                  /* data to pass to function */
);
```

pthread_cleanup_pop—uninstall cleanup handler

```
#include <pthread.h>

void pthread_cleanup_pop(
    int execute                /* execute handler? */
);
```

When cancellation occurs, the cleanup handler is called. If there's more than one, they're all called, in the reverse order of how they were pushed. Each function is popped after it's called, which is fine, since the thread will be gone.

These functions must be paired, and they must be at the same C or C++ block level. If you don't do this, you'll probably get very strange *compile-time* errors, as the functions are usually implemented as macros with embedded braces (like `EC_CLEANUP_BGN` and `EC_CLEANUP_END`; see Section 1.4.2).

Depending on how you've done things, it may make sense to call the cleanup handler even when the thread exits normally. To make that convenient, you can make the `execute` argument to `pthread_cleanup_pop` true.

Here's the improved version of `thread_func`. Note that I explicitly initialized `p` to `NULL` to make sure its value is always valid for `free`.

```
static void cleanup_handler(void *arg)
{
    free(arg);
    (void)pthread_mutex_unlock(&mtx);
}

static void *thread_func(void *arg)
{
    struct node *p = NULL;

    pthread_cleanup_push(cleanup_handler, p);
    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    pthread_cleanup_pop(false);
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return (void *)false;
EC_CLEANUP_END
}
```

Now we should be OK. Note that the whole complication of dealing with cancellation could have been avoided if I chose some other way to terminate the thread once the initial thread had stopped putting nodes on the queue. That's something to consider in your own applications—don't use cancellation when a less ruthless action will do the job. On the other hand, if the thread to be terminated is blocked (in `read`, say) and you can't very easily get it unblocked, thread cancellation may be the best choice. It's certainly a better choice than a signal, which also can unblock a system call; as we shall see in Chapter 9, signals have much worse side effects.

5.17.6 Threads vs. Processes

You're probably wondering when you should use threads vs. processes. Generally, you use threads when you want concurrent processing on the same complex data structures. Three common situations are:

- You want the user interface to be active while other computations are going on in the background. An example would be background printing or background page formatting in a word-processing program that allows the document to be viewed and edited concurrently with those background operations.
- You want to organize the algorithm to take advantage of a multiprocessing computer—one with multiple CPUs. On such a system, the UNIX scheduler can automatically assign different threads to different CPUs.
- You need to deal with several kinds of events that cause systems calls (e.g., `read`, `waitpid`, `msgrcv`) to block. This is the subject of the next section.

Processes would be used for less tightly coupled applications, where some data is passed between the processes, but there's no need for the same data structure to be manipulated directly. (With some trouble, it is possible to share a data structure using shared memory, as we'll see in Chapter 7.) Separate processes have their own effective user ID, file descriptors, global variables, etc., whereas separate threads don't. Processes can more easily be developed, tested, and debugged separately, and there's much less need for locking, so much less likelihood of undetected race conditions or deadlock. Another way to draw the comparison is to say that processes are for the big pieces of the applications, whereas threads provide more fine-grained concurrency.

Warning: As of this writing, the thread packages commonly shipped with Linux and FreeBSD don't adhere to the POSIX standard, chiefly because they implement a thread either too weakly (doing everything in user space) or too strongly (using a process for each thread). The main problem with the former is that a blocking system call, like `msgrcv`, blocks all threads, not just the one containing the call. The problem with the latter is that some systems calls like `waitpid` don't work right because they're in the wrong process (only the parent may wait for a child process). Another irritant is that you may have to find, compile, and install the thread package you want yourself if the package that came with the system is inadequate.

Look for the newest Linux thread implementation, called Native POSIX Thread Library for Linux (NPTL), which is now finding its way into Linux systems. It fixes all of the important POSIX-compliance bugs.

5.18 The Blocking Problem

We've encountered several system calls that can block, like `read`, `write`, `pthread_cond_wait`, and `waitpid`, and there are lots more in this book, especially in Chapters 7 and 8. One of the most difficult problems with UNIX programming is that your application may have to block in more than one system call because you don't know what's going to happen next. I call this the blocking problem.

For blocking system calls that take a file descriptor, like `read` and `write`, you can use a single system call, `select` or `poll` (Section 4.2), to block until one or more file descriptors are ready. But that doesn't help in general because lots of things you wait for, such as processes, signals, messages, semaphores, and condition variables, aren't associated with file descriptors. Yes, you can notify a thread blocked in `select` or `poll` when, say, a message arrives by writing to a pipe set up just for that purpose (as we'll see in the next section), but that doesn't help with the fact that `msgrcv` or `mq_receive` was itself blocked waiting for the message in the first place. There's no way to tie a message-waiting system call directly to a file descriptor. It's not a *notification* problem, it's a *blocking* problem.

5.18.1 Solutions Using Processes and Threads

Historically, the solution to the blocking problem was to create a child process to block.[24] When whatever it's blocking on unblocks, it then writes to a pipe to indicate to its parent process that an event has occurred. Pipes are a good choice because they're easy to set up (as we'll see in the next chapter) and, as they use file descriptors, the parent can incorporate them into a `select` or `poll`. What this does, in effect, is to transform the parent from blocking on the non-file-descriptor event to blocking on a file descriptor. Transform everything, use `select` or `poll` in the parent, and you're golden.

But processes are very heavyweight objects in UNIX—expensive to create, expensive to schedule, and in limited supply. Also, it's cumbersome for two processes to share the same data structures. It would be nice if the event could just be queued up when it occurs and then examined whenever it's convenient for the

24. This technique was invented by my mother around 1953. At the grocery store, she would have one child wait on the deli line and one on the fish line while she shopped for goods that didn't require a line. Occasionally even she got blocked, though, so she produced three more children.

parent. Setting up an event queue between processes is possible using shared memory and semaphores, but interprocess semaphores may be too slow, especially if the events are occurring rapidly.

Another problem with separate processes is that certain objects, like file descriptors, can't be passed to an existing process (not portably, anyway). They can be passed only through inheritance; therefore, if process A is in charge of blocking on I/O and is already running, process B that just opened a network connection can't get process A to wait on the newly acquired file descriptor.

A newer solution for the blocking problem is to use POSIX Threads. One simple approach is to create a new thread for each object on which you want to block (e.g., message queue, file-descriptor set, semaphore). Each thread simply issues an appropriate blocking system call (`O_NONBLOCK` clear). When the system call returns, the thread adds an event to a shared queue (using a mutex for protection) and then goes right back into the blocking system call. The main thread then has only one thing to block on: the presence of an event on the queue. The blocking threads use a condition variable to signal the main thread when the queue becomes nonempty, exactly as in the example in Section 5.17.4.

5.18.2 Unified Event Manager Prototype

I'll show a generalized approach to using threads to solve the blocking problem that I call a Unified Event Manager. It's a collection of library functions that any application can use. An application can register an event, which causes the library to create a thread to block. The application is organized around waiting on a single event queue. When an event shows up, the application takes it off the queue, processes it, and goes back to waiting.

I won't show all the code in this book, but it's on the Web site if you want to see it. It's a *prototype* because it's not efficient enough for critical applications and because it uses the same "ec" error-checking approach that all the examples in this book use, and that is itself a prototype.

We start with an enumeration for almost all events that can be waited for in UNIX:

```
enum UEM_TYPE {
    UEM_SVMSG,      /* System V message */
    UEM_PXMSG,      /* POSIX message */
    UEM_SVSEM,      /* System V semaphore */
    UEM_PXSEM,      /* POSIX semaphore */
    UEM_FD_READ,    /* file-descriptor set - read */
```

```
    UEM_FD_WRITE,   /* file-descriptor set - write */
    UEM_FD_ERROR,   /* file-descriptor set - error */
    UEM_SIG,        /* signal */
    UEM_PROCESS,    /* process */
    UEM_HEARTBEAT,  /* heartbeat */
    UEM_NONE        /* none */
};
```

The last two require some explanation: `UEM_HEARTBEAT` is a way to just get a periodic event at certain intervals, and `UEM_NONE` is there because it's always a good idea to have a value that you can use to indicate that something's empty.

When we register an event, we need a structure to keep track of what we'll need for the blocking system call. For example, to issue a `select`, we'll need the file-descriptor set, so this structure holds all of this kind of data for each `UEM_TYPE`:

```
struct uem_reg {
    enum UEM_TYPE ur_type;          /* type of registration */
    pthread_t ur_tid;               /* thread ID */
    union {
        int ur_mqid;                /* System V message-queue ID */
        struct {
            int s_semid;            /* System V semaphore-set ID */
            struct sembuf *s_sops;  /* semaphore operations */
        } ur_svsem;
#ifdef POSIX_IPC
        mqd_t ur_mqd;               /* POSIX message-queue descriptor */
        sem_t *ur_sem;              /* POSIX semaphore */
#endif
        int ur_signum;              /* signal number */
        pid_t ur_pid;               /* process ID */
        long ur_usecs;              /* microseconds (for heartbeat) */
        fd_set ur_fdset;            /* file-descriptor set */
    } ur_resource;
    void *ur_data;                  /* data to be queued with event */
    size_t ur_size;                 /* size (used for various purposes) */
};
```

The macro `POSIX_IPC` isn't a standard macro but one that we'll use in this prototype. It's set from the real feature-test macros in a very complicated way that's explained in Section 1.5.4. We need it because Linux and FreeBSD don't as yet support POSIX IPC.

When an application wants to register an event type, it calls a function of the form `uem_register_`*E,* where *E* is an abbreviation for the event type. For example, here's the call to register to wait for a process to terminate:

```
ec_false( uem_register_process(pid, NULL) )
```

This is instead of issuing a `waitpid` directly, which would block. When process `pid` terminates, an event containing its exit status will be put on the queue, and the application that registered the event can then get the status. We'll see those details shortly.

Here's the code for `uem_register_process`:

```
bool uem_register_process(pid_t pid, void *data)
{
    struct uem_reg *p;

    ec_null( p = new_reg() )
    p->ur_type = UEM_PROCESS;
    p->ur_resource.ur_pid = pid;
    p->ur_size = 0;
    p->ur_data = data;
    ec_rv( pthread_create(&p->ur_tid, NULL, thread_process, p) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The function `new_reg`, used by all the `uem_register_` *E* calls, just allocates a structure. We put it in a separate function in case there's some common initialization to do, which there isn't with the current design.

```
static struct uem_reg *new_reg(void)
{
    struct uem_reg *p;

    ec_null( p = calloc(1, sizeof(struct uem_reg)) )
    return p;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

The registration info is just passed onto the thread function, which looks like this:

```
static void *thread_process(void *arg)
{
    struct uem_event *e = NULL;

    pthread_cleanup_push(cleanup_handler, e);
    ec_null( e = calloc(1, sizeof(struct uem_event)) )
```

```
    e->ue_reg = (struct uem_reg *)arg;
    if (waitpid(e->ue_reg->ur_resource.ur_pid, &e->ue_result, 0) == -1)
        e->ue_errno = errno;
    ec_false( queue_event(e) )
    pthread_cleanup_pop(false);
    return NULL;

EC_CLEANUP_BGN
    uem_free(e);
    EC_FLUSH("thread_process")
    return NULL;
EC_CLEANUP_END
}
```

This function first pushes a cleanup handler (explained in Section 5.17.5). Then it allocates an event structure to be queued when the event occurs (`waitpid` returning, in this case). Here's the event structure used by all the threads:

```
struct uem_event {
    struct uem_reg *ue_reg;
    void *ue_buf;
    ssize_t ue_result;
    int ue_errno;
    struct uem_event *ue_next;
};
```

Note that it points back to the registration, which contains data common to all the events of this type. The `ue_buf` member is in case data has to be returned (such as a message), but there's none in this case. We do have the status, which we put in the `ue_result` member. If an error occurred, `errno` goes into the `ue_errno` member; the application that gets this event needs to check that member against zero to see if an error was returned by the function that waited. The `ue_next` member is so that the `uem_event` structures can be linked into an event queue.

The cleanup handler is just like the one in Section 5.17.5 except it calls `uem_free` (not shown here) to free the event structure if one was allocated:

```
static void cleanup_handler(void *arg)
{
    (void)uem_free((struct uem_event *)arg);
}
```

An application that removes an event from the queue is also responsible for calling `uem_free`.

The actual work of putting an event on the queue is done by `queue_event`, which the thread calls when `waitpid` returns. Note that the event is queued even if `waitpid` reported an error, which is how the application finds out about such errors.

```
static pthread_mutex_t uem_mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t uem_cond_event = PTHREAD_COND_INITIALIZER;
static struct uem_event *event_head;

static bool queue_event(struct uem_event *e)
{
    struct uem_event *cur;

    ec_rv( pthread_mutex_lock(&uem_mtx) )
    if (event_head == NULL)
        event_head = e;
    else {
        for (cur = event_head; cur->ue_next != NULL; cur = cur->ue_next)
            /* queue same error only once */
            if (e->ue_errno != 0 &&
              cur->ue_reg->ur_type == e->ue_reg->ur_type &&
              cur->ue_errno == e->ue_errno) {
                ec_rv( pthread_mutex_unlock(&uem_mtx) )
                uem_free(e);
                return true;
            }
        cur->ue_next = e;
    }
    ec_rv( pthread_cond_signal(&uem_cond_event) )
    ec_rv( pthread_mutex_unlock(&uem_mtx) )
    return true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&uem_mtx);
    return false;
EC_CLEANUP_END
}
```

This function follows the pattern from Section 5.17.4 for using a condition variable for signaling an event. However, an error event (`ue_errno` nonzero) could be generated repeatedly if the thread that calls `queue_event` keeps getting an error. For example, if the process ID passed to `waitpid` is invalid, `waitpid` will keep returning until the thread is cancelled. It's bad enough that that wastes CPU time, but we certainly don't want to fill the event queue with thousands of events all reporting the same thing. So, before queuing an error event, we make sure one just like it isn't already there. (This isn't the best approach, but this is a prototype, right?)

If the event is to be queued, the line

```
cur->ue_next = e;
```

queues it and then we signal the condition, unlock the mutex, and return.

That's basically the whole library except that there are a collection of registration functions like `uem_register_process`, which I won't show here because they all do pretty much the same thing, varying only in the system call that blocks. That is, `uem_register_svmsg` starts a thread that contains a call to `msgrcv`, `uem_register_pxmsg` starts a thread that contains a call to `mq_receive`, and so on. They all use `queue_event`.

The application that uses the library looks something like this:

```
struct uem_event *e;
...
ec_false( uem_register_process(pid, NULL) )
ec_false( uem_register_pxmsg(mqd, NULL) )
...
while (true) {
    ec_null( e = uem_wait() )
    if (e->ue_errno != 0)
        ... /* display error */
    else
        switch (e->ue_reg->ur_type) {
        case UEM_PXMSG:
            ... /* process received message */
            break;
        case UEM_PROCESS:
            ... /* process status from terminated process */
            break;
        ...
        }
}
```

The most important thing here is that the application blocks in exactly one place, the call to `uem_wait`, no matter how many different kinds of events need to be handled. Here's the code for `uem_wait`; note that, like `queue_event`, it follows the conditional-variable pattern in Section 5.17.4:

```
struct uem_event *uem_wait(void)
{
    struct uem_event *e = NULL;
```

```
    ec_rv( pthread_mutex_lock(&uem_mtx) )
    while (event_head == NULL)
        ec_rv( pthread_cond_wait(&uem_cond_event, &uem_mtx) )
    e = event_head;
    event_head = event_head->ue_next;
    ec_rv( pthread_mutex_unlock(&uem_mtx) )
    return e;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&uem_mtx);
    return NULL;
EC_CLEANUP_END
}
```

When the predicate is true (event present on queue), it removes the event from the queue and returns a pointer to it. It's the responsibility of the caller to free the memory with a call to `uem_free`, as I mentioned earlier.

Except for some additional bookkeeping details and the rest of the registration functions, that's the whole system. It solves the blocking problem!

Because it's so reliant on threads, this approach is only as good as the underlying thread implementation. Threads have to be fast, lightweight, and plentiful, and they have to follow the POSIX standard rigorously. Otherwise, there will be too much overhead, valuable system resources will be consumed, and there will be too many subtle bugs in the application that, because of all the multithreading, will be very hard to find.

If you're interested, try to rewrite the uem package using processes instead of threads. You'll find it difficult to write and extremely difficult to make efficient. But the effort should help you appreciate why threads are so important.

Exercises

5.1. Correct the memory-leak problems in `setenv` and `unsetenv` as suggested at the end of Section 5.2.

5.2. Rewrite the environment-manipulation functions in Section 5.2 to treat exported variables as the standard shell does. That is, an updated value for a variable is exported only if it's specifically declared to be exported in a function named, say, `env_export`. Give some thought to whether and when `environ` will be updated

and whether the existing `getenv` function can be used as is or whether it needs to be replaced.

5.3. Write a program that scans its arguments for assignments of the form *variable=value,* updates the environment appropriately, and then executes the program specified by the first nonassignment argument. The other nonassignment arguments become arguments to the invoked program. Don't use `fork`.

5.4. Write a function `execlp2` that's analogous to `execvp2` in Section 5.3. Use the Standard C variable-argument facilities (`va_arg`, etc.).

5.5. Enhance the `exec_path` function in Section 5.3 to support the `#!` feature discussed in that section.

5.6. Design and implement two functions, `execvx` and `execlx`, to replace the six `exec` system calls, as suggested by the footnote in Section 5.3. You can use as many of the `exec` system calls as you wish in your implementation.

5.7. Explain in your own words (perhaps with a diagram) how `fork` and `exec` are typically implemented. (You'll need to do some research; see, for example, [Bac1986], [McK1996], [Mau2001], or [Bov2001].) Then show how `posix_spawn` can be implemented more efficiently. Don't write the code—pseudo code or a clear step-by-step algorithm will do.

5.8. Research the semantics of `posix_spawn` (see [SUS2002]) and then implement it in terms of `fork` and `exec`, as suggested at the end of Section 5.5. To start with, skip both attributes and actions. Then do actions, and finally attributes. To do the whole job you'll need to implement the associated system calls (e.g., `posix_spawn_file_actions_init`). This is an extremely useful exercise, even if you don't care about realtime systems.

5.9. Design and run an experiment to measure the CPU time used by `fork`. (You may want to use `timestart` and `timestop` from Section 1.7.2.) If you have access to them, try different versions of UNIX and different hardware. Compare the time for `fork` with the time for `vfork`, if your systems support it. Ditto for `posix_spawn`.

5.10. Investigate the equivalent system calls to `exec` and `fork` that are provided in other operating systems, such as VMS, OS/390, Windows, and MacOS. Compare their features and summarize their advantages and disadvantages.

5.11. In Section 5.17.3, it's possible for `x` to be incremented by another thread between reading with `get_x(0)` and incrementing with `get_x(1)`, which might result in `x` being too big. Fix the problem by rewriting the example to use a single function call that both tests and increments `x`.

5.12. Write `fileview`, an interactive full-screen application (Section 4.8) that divides the screen with a horizontal line into two parts. An "s" command prompts for a search string, searches the standard include files (at least) for files that contain it on one or more lines, and displays the matching pathnames in the upper part. While they're being displayed or afterwards, a "v" command shows the contents of a selected file in the lower part with the matched strings highlighted. Choose two keys that can be used to scroll the upper part up and down, and two other keys for the lower part. To make the "v" command easy to implement, number the pathnames in the upper part and arrange for the "v" command to prompt for the file by its number. Note the word "While" in the phrase "While they're being displayed"; it implies that you'll have to use multithreading and, as Curses isn't necessarily thread-safe, you'll have to protect access to it with a mutex. There's also a "q" command that quits the application.

5.13. Explain how you could implement `fileview` (Exercise 5.12) without multithreading, using processes instead. You may have to explore Chapter 7 first. If you don't think you can do it without multithreading, explain why not. (As you would be trying to prove a negative statement, your explanation will have to be very persuasive.)

5.14. Write a program to display as many as the process attributes listed in Appendix A as you can, limiting yourself for now to those discussed in the first five chapters of this book. Later, you can extend your program to include them all. If there are any you can't display, explain why not. To make the output interesting, execute some system calls at the start to open some files, set some signal actions, and so on.

This page intentionally left blank