7

Process Environment

7.1 Introduction

Before looking at the process control primitives in the next chapter, we need to examine the environment of a single process. In this chapter, we'll see how the `main` function is called when the program is executed, how command-line arguments are passed to the new program, what the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables, and various ways for the process to terminate. Additionally, we'll look at the `longjmp` and `setjmp` functions and their interaction with the stack. We finish the chapter by examining the resource limits of a process.

7.2 `main` Function

A C program starts execution with a function called `main`. The prototype for the `main` function is

```
int main(int argc, char *argv[]);
```

where *argc* is the number of command-line arguments, and *argv* is an array of pointers to the arguments. We describe these arguments in Section 7.4.

When a C program is executed by the kernel—by one of the `exec` functions, which we describe in Section 8.10—a special start-up routine is called before the `main` function is called. The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from the kernel—the command-line arguments and the environment—and sets things up so that the `main` function is called as shown earlier.

7.3 Process Termination

There are eight ways for a process to terminate. Normal termination occurs in five ways:

1. Return from `main`
2. Calling `exit`
3. Calling `_exit` or `_Exit`
4. Return of the last thread from its start routine (Section 11.5)
5. Calling `pthread_exit` (Section 11.5) from the last thread

Abnormal termination occurs in three ways:

6. Calling `abort` (Section 10.17)
7. Receipt of a signal (Section 10.2)
8. Response of the last thread to a cancellation request (Sections 11.5 and 12.7)

> For now, we'll ignore the three termination methods specific to threads until we discuss threads in Chapters 11 and 12.

The start-up routine that we mentioned in the previous section is also written so that if the `main` function returns, the `exit` function is called. If the start-up routine were coded in C (it is often coded in assembly language) the call to `main` could look like

```
exit(main(argc, argv));
```

Exit Functions

Three functions terminate a program normally: `_exit` and `_Exit`, which return to the kernel immediately, and `exit`, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

We'll discuss the effect of these three functions on other processes, such as the children and the parent of the terminating process, in Section 8.5.

> The reason for the different headers is that `exit` and `_Exit` are specified by ISO C, whereas `_exit` is specified by POSIX.1.

Historically, the `exit` function has always performed a clean shutdown of the standard I/O library: the `fclose` function is called for all open streams. Recall from Section 5.5 that this causes all buffered output data to be flushed (written to the file).

All three exit functions expect a single integer argument, which we call the *exit status*. Most UNIX System shells provide a way to examine the exit status of a process. If (a) any of these functions is called without an exit status, (b) `main` does a `return` without a return value, or (c) the `main` function is not declared to return an integer, the exit status of the process is undefined. However, if the return type of `main` is an integer and `main` "falls off the end" (an implicit return), the exit status of the process is 0.

> This behavior is new with the 1999 version of the ISO C standard. Historically, the exit status was undefined if the end of the `main` function was reached without an explicit `return` statement or a call to the `exit` function.

Returning an integer value from the `main` function is equivalent to calling `exit` with the same value. Thus

```
exit(0);
```

is the same as

```
return(0);
```

from the `main` function.

Example

The program in Figure 7.1 is the classic "hello, world" example.

```
#include    <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Figure 7.1 Classic C program

When we compile and run the program in Figure 7.1, we see that the exit code is random. If we compile the same program on different systems, we are likely to get different exit codes, depending on the contents of the stack and register contents at the time that the `main` function returns:

```
$ gcc hello.c
$ ./a.out
hello, world
$ echo $?                          print the exit status
13
```

Now if we enable the 1999 ISO C compiler extensions, we see that the exit code changes:

```
$ gcc -std=c99 hello.c          enable gcc's 1999 ISO C extensions
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
hello, world
$ echo $?                       print the exit status
0
```

Note the compiler warning when we enable the 1999 ISO C extensions. This warning is printed because the type of the `main` function is not explicitly declared to be an integer. If we were to add this declaration, the message would go away. However, if we were to enable all recommended warnings from the compiler (with the `-Wall` flag), then we would see a warning message something like "control reaches end of nonvoid function."

The declaration of `main` as returning an integer and the use of `exit` instead of `return` produces needless warnings from some compilers and the `lint`(1) program. The problem is that these compilers don't know that an `exit` from `main` is the same as a `return`. One way around these warnings, which become annoying after a while, is to use `return` instead of `exit` from `main`. But doing this prevents us from using the UNIX System's `grep` utility to locate all calls to `exit` from a program. Another solution is to declare `main` as returning `void`, instead of `int`, and continue calling `exit`. This gets rid of the compiler warning but doesn't look right (especially in a programming text), and can generate other compiler warnings, since the return type of `main` is supposed to be a signed integer. In this text, we show `main` as returning an integer, since that is the definition specified by both ISO C and POSIX.1.

Different compilers vary in the verbosity of their warnings. Note that the GNU C compiler usually doesn't emit these extraneous compiler warnings unless additional warning options are used.

□

In the next chapter, we'll see how any process can cause a program to be executed, wait for the process to complete, and then fetch its exit status.

`atexit` Function

With ISO C, a process can register at least 32 functions that are automatically called by `exit`. These are called *exit handlers* and are registered by calling the `atexit` function.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

This declaration says that we pass the address of a function as the argument to `atexit`. When this function is called, it is not passed any arguments and is not expected to return a value. The `exit` function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

These exit handlers first appeared in the ANSI C Standard in 1989. Systems that predate ANSI C, such as SVR3 and 4.3BSD, did not provide these exit handlers.

ISO C requires that systems support at least 32 exit handlers, but implementations often support more (see Figure 2.15). The `sysconf` function can be used to determine the maximum number of exit handlers supported by a given platform, as illustrated in Figure 2.14.

With ISO C and POSIX.1, `exit` first calls the exit handlers and then closes (via `fclose`) all open streams. POSIX.1 extends the ISO C standard by specifying that any exit handlers installed will be cleared if the program calls any of the `exec` family of functions. Figure 7.2 summarizes how a C program is started and the various ways it can terminate.

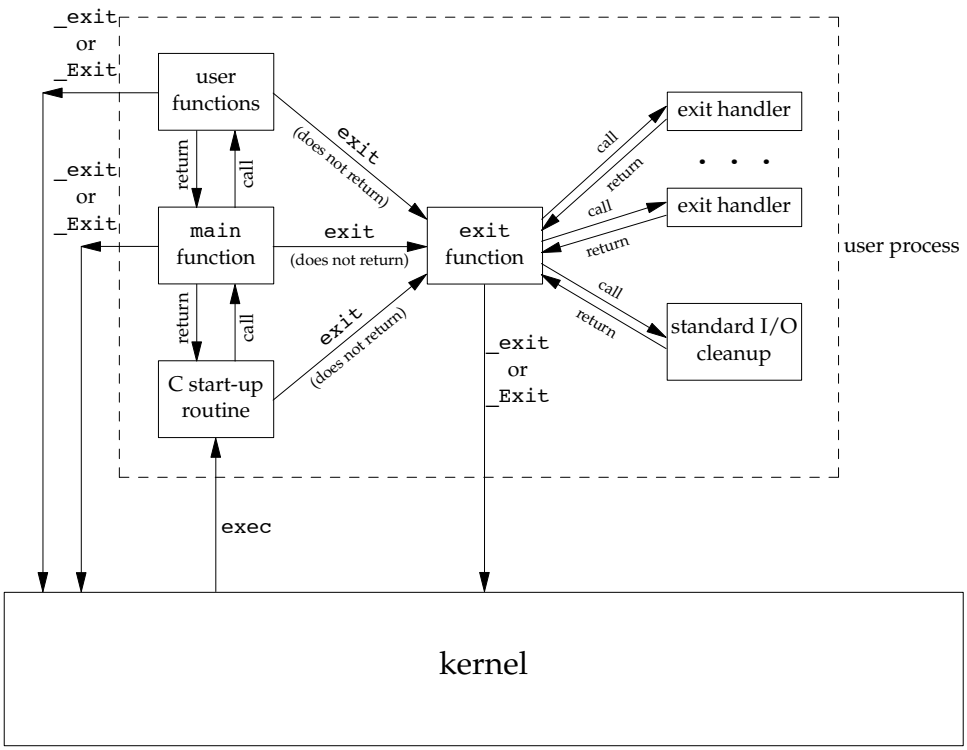


Figure 7.2 How a C program is started and how it terminates

The only way a program can be executed by the kernel is if one of the `exec` functions is called. The only way a process can voluntarily terminate is if `_exit` or `_Exit` is called, either explicitly or implicitly (by calling `exit`). A process can also be involuntarily terminated by a signal (not shown in Figure 7.2).

Example

The program in Figure 7.3 demonstrates the use of the `atexit` function.

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

Figure 7.3 Example of exit handlers

Executing the program in Figure 7.3 yields

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

An exit handler is called once for each time it is registered. In Figure 7.3, the first exit handler is registered twice, so it is called two times. Note that we don't call `exit`; instead, we return from `main`. □

7.4 Command-Line Arguments

When a program is executed, the process that does the `exec` can pass command-line arguments to the new program. This is part of the normal operation of the UNIX system shells. We have already seen this in many of the examples from earlier chapters.

Example

The program in Figure 7.4 echoes all its command-line arguments to standard output. Note that the normal `echo`(1) program doesn't echo the zeroth argument.

```
#include "apue.h"

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

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Figure 7.4 Echo all command-line arguments to standard output

If we compile this program and name the executable `echoarg`, we have

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

We are guaranteed by both ISO C and POSIX.1 that `argv[argc]` is a null pointer. This lets us alternatively code the argument-processing loop as

```
for (i = 0; argv[i] != NULL; i++)
```

□

7.5 Environment List

Each program is also passed an *environment list*. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable `environ`:

```
extern char **environ;
```

For example, if the environment consisted of five strings, it could look like Figure 7.5. Here we explicitly show the null bytes at the end of each string. We'll call `environ` the

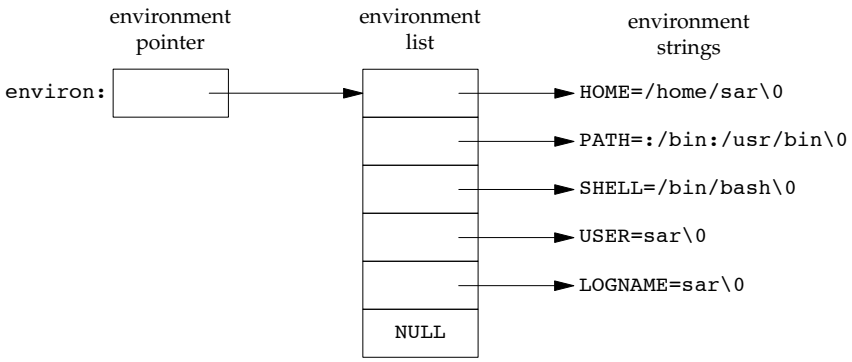


Figure 7.5 Environment consisting of five C character strings

environment pointer, the array of pointers the environment list, and the strings they point to the *environment strings*.

By convention, the environment consists of

name=value

strings, as shown in Figure 7.5. Most predefined names are entirely uppercase, but this is only a convention.

Historically, most UNIX systems have provided a third argument to the `main` function that is the address of the environment list:

```
int main(int argc, char *argv[], char *envp[]);
```

Because ISO C specifies that the `main` function be written with two arguments, and because this third argument provides no benefit over the global variable `environ`, POSIX.1 specifies that `environ` should be used instead of the (possible) third argument. Access to specific environment variables is normally through the `getenv` and `putenv` functions, described in Section 7.9, instead of through the `environ` variable. But to go through the entire environment, the `environ` pointer must be used.

7.6 Memory Layout of a C Program

Historically, a C program has been composed of the following pieces:

- Text segment, consisting of the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

- Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

```
int   maxcount = 99;
```

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

- Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

```
long   sum[1000];
```

  appearing outside any function causes this variable to be stored in the uninitialized data segment.

- Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

- Heap, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.

Figure 7.6 shows the typical arrangement of these segments. This is a logical picture of how a program looks; there is no requirement that a given implementation arrange its memory in this fashion. Nevertheless, this gives us a typical arrangement to describe. With Linux on a 32-bit Intel x86 processor, the text segment starts at location `0x08048000`, and the bottom of the stack starts just below `0xC0000000`. (The stack grows from higher-numbered addresses to lower-numbered addresses on this particular architecture.) The unused virtual address space between the top of the heap and the top of the stack is large.

> Several more segment types exist in an `a.out`, containing the symbol table, debugging information, linkage tables for dynamic shared libraries, and the like. These additional sections don't get loaded as part of the program's image executed by a process.

Note from Figure 7.6 that the contents of the uninitialized data segment are not stored in the program file on disk, because the kernel sets the contents to 0 before the program starts running. The only portions of the program that need to be saved in the program file are the text segment and the initialized data.

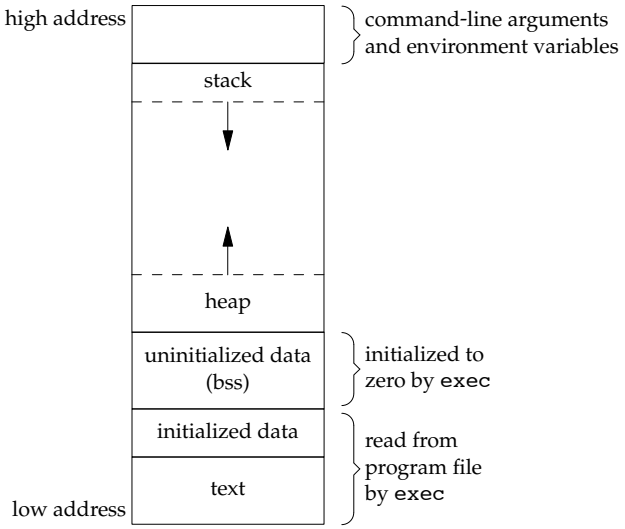


Figure 7.6 Typical memory arrangement

The `size`(1) command reports the sizes (in bytes) of the text, data, and bss segments. For example:

```
$ size /usr/bin/cc /bin/sh
   text    data     bss     dec     hex  filename
 346919    3576    6680  357175   57337  /usr/bin/cc
 102134    1776   11272  115182   1c1ee  /bin/sh
```

The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively.

7.7 Shared Libraries

Most UNIX systems today support shared libraries. Arnold [1986] describes an early implementation under System V, and Gingell et al. [1987] describe a different implementation under SunOS. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit every program that uses the library (assuming that the number and type of arguments haven't changed).

Different systems provide different ways for a program to say that it wants to use or not use the shared libraries. Options for the `cc`(1) and `ld`(1) commands are typical. As

an example of the size differences, the following executable file—the classic `hello.c` program—was first created without shared libraries:

```
$ gcc -static hello1.c          prevent gcc from using shared libraries
$ ls -l a.out
-rwxr-xr-x  1 sar     879443 Sep 2 10:39 a.out
$ size a.out
   text    data     bss     dec     hex  filename
 787775    6128   11272  805175   c4937  a.out
```

If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ gcc hello1.c                  gcc defaults to use shared libraries
$ ls -l a.out
-rwxr-xr-x  1 sar       8378 Sep 2 10:39 a.out
$ size a.out
   text    data     bss     dec     hex  filename
   1176     504      16    1696     6a0  a.out
```

7.8 Memory Allocation

ISO C specifies three functions for memory allocation:

1. `malloc`, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
2. `calloc`, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
3. `realloc`, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

                        All three return: non-null pointer if OK, NULL on error

void free(void *ptr);
```

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. For example, if the most restrictive alignment requirement on a particular system requires that `doubles` must start at memory locations that are multiples of 8, then all pointers returned by these three functions would be so aligned.

Because the three `alloc` functions return a generic `void *` pointer, if we `#include <stdlib.h>` (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type. The default return value for undeclared functions is `int`, so using a cast without the proper function declaration could hide an error on systems where the size of type `int` differs from the size of a function's return value (a pointer in this case).

The function `free` causes the space pointed to by *ptr* to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three `alloc` functions.

The `realloc` function lets us change the size of a previously allocated area. (The most common usage is to increase an area's size.) For example, if we allocate room for 512 elements in an array that we fill in at runtime but later find that we need more room, we can call `realloc`. If there is room beyond the end of the existing region for the requested space, then `realloc` simply allocates this additional area at the end and returns the same pointer that we passed it. But if there isn't room, `realloc` allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area. Because the area may move, we shouldn't have any pointers into this area. Exercise 4.16 and Figure C.3 show the use of `realloc` with `getcwd` to handle a pathname of any length. Figure 17.27 shows an example that uses `realloc` to avoid arrays with fixed, compile-time sizes.

Note that the final argument to `realloc` is the new size of the region, not the difference between the old and new sizes. As a special case, if *ptr* is a null pointer, `realloc` behaves like `malloc` and allocates a region of the specified *newsize*.

> Older versions of these routines allowed us to `realloc` a block that we had `free`d since the last call to `malloc`, `realloc`, or `calloc`. This trick dates back to Version 7 and exploited the search strategy of `malloc` to perform storage compaction. Solaris still supports this feature, but many other platforms do not. This feature is deprecated and should not be used.

The allocation routines are usually implemented with the `sbrk`(2) system call. This system call expands (or contracts) the heap of the process. (Refer to Figure 7.6.) A sample implementation of `malloc` and `free` is given in Section 8.7 of Kernighan and Ritchie [1988].

Although `sbrk` can expand or contract the memory of a process, most versions of `malloc` and `free` never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; instead, that space is kept in the `malloc` pool.

Most implementations allocate more space than requested and use the additional space for record keeping—the size of the block, a pointer to the next allocated block, and the like. As a consequence, writing past the end or before the start of an allocated area could overwrite this record-keeping information in another block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later.

Writing past the end or before the beginning of a dynamically allocated buffer can corrupt more than internal record-keeping information. The memory before and after a dynamically allocated buffer can potentially be used for other dynamically allocated

objects. These objects can be unrelated to the code corrupting them, making it even more difficult to find the source of the corruption.

Other possible errors that can be fatal are freeing a block that was already freed and calling `free` with a pointer that was not obtained from one of the three `alloc` functions. If a process calls `malloc` but forgets to call `free`, its memory usage will continually increase; this is called leakage. If we do not call `free` to return unused space, the size of a process's address space will slowly increase until no free space is left. During this time, performance can degrade from excess paging overhead.

Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three `alloc` functions or `free` is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

> FreeBSD, Mac OS X, and Linux support additional debugging through the setting of environment variables. In addition, options can be passed to the FreeBSD library through the symbolic link `/etc/malloc.conf`.

Alternate Memory Allocators

Many replacements for `malloc` and `free` are available. Some systems already include libraries providing alternative memory allocator implementations. Other systems provide only the standard allocator, leaving it up to software developers to download alternatives, if desired. We discuss some of the alternatives here.

`libmalloc`

SVR4-based systems, such as Solaris, include the `libmalloc` library, which provides a set of interfaces matching the ISO C memory allocation functions. The `libmalloc` library includes `mallopt`, a function that allows a process to set certain variables that control the operation of the storage allocator. A function called `mallinfo` is also available to provide statistics on the memory allocator.

`vmalloc`

Vo [1996] describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to `vmalloc`, the library provides emulations of the ISO C memory allocation functions.

`quick-fit`

Historically, the standard `malloc` algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Weinstock and Wulf [1988] describe the algorithm, which is based on splitting up memory into buffers of various sizes and maintaining unused buffers on different free lists, depending on the buffer sizes. Most modern allocators are based on quick-fit.

jemalloc

The jemalloc implementation of the malloc family of library functions is the default memory allocator in FreeBSD 8.0. It was designed to scale well when used with multithreaded applications running on multiprocessor systems. Evans [2006] describes the implementation and evaluates its performance.

TCMalloc

TCMalloc was designed as a replacement for the malloc family of functions to provide high performance, scalability, and memory efficiency. It uses thread-local caches to avoid locking overhead when allocating buffers from and releasing buffers to the cache. It also has a heap checker and a heap profiler built in to aid in debugging and analyzing dynamic memory usage. The TCMalloc library is available as open source from Google. It is briefly described by Ghemawat and Menage [2005].

alloca Function

One additional function is also worth mentioning. The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

All four platforms discussed in this text provide the alloca function.

7.9 Environment Variables

As we mentioned earlier, the environment strings are usually of the form

name=*value*

The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login; others are left for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. If we set the environment variable MAILPATH, for example, it tells the Bourne shell, GNU Bourne-again shell, and Korn shell where to look for mail.

ISO C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns: pointer to *value* associated with *name*, NULL if not found

Note that this function returns a pointer to the *value* of a *name=value* string. We should always use `getenv` to fetch a specific value from the environment, instead of accessing `environ` directly.

Some environment variables are defined by POSIX.1 in the Single UNIX Specification, whereas others are defined only if the XSI option is supported. Figure 7.7 lists the environment variables defined by the Single UNIX Specification and notes which implementations support the variables. Any environment variable defined by POSIX.1 is marked with •; otherwise, it is part of the XSI option. Many additional implementation-dependent environment variables are used in the four implementations described in this book. Note that ISO C doesn't define any environment variables.

Variable	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
COLUMNS	•	•	•	•	•	terminal width
DATEMSK	XSI		•	•	•	`getdate`(3) template file pathname
HOME	•	•	•	•	•	home directory
LANG	•	•	•	•	•	name of locale
LC_ALL	•	•	•	•	•	name of locale
LC_COLLATE	•	•	•	•	•	name of locale for collation
LC_CTYPE	•	•	•	•	•	name of locale for character classification
LC_MESSAGES	•	•	•	•	•	name of locale for messages
LC_MONETARY	•	•	•	•	•	name of locale for monetary editing
LC_NUMERIC	•	•	•	•	•	name of locale for numeric editing
LC_TIME	•	•	•	•	•	name of locale for date/time formatting
LINES	•	•	•	•	•	terminal height
LOGNAME	•	•	•	•	•	login name
MSGVERB	XSI	•	•	•	•	`fmtmsg`(3) message components to process
NLSPATH	•	•	•	•	•	sequence of templates for message catalogs
PATH	•	•	•	•	•	list of path prefixes to search for executable file
PWD	•	•	•	•	•	absolute pathname of current working directory
SHELL	•	•	•	•	•	name of user's preferred shell
TERM	•	•	•	•	•	terminal type
TMPDIR	•	•	•	•	•	pathname of directory for creating temporary files
TZ	•	•	•	•	•	time zone information

Figure 7.7 Environment variables defined in the Single UNIX Specification

In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. (In the next chapter, we'll see that we can affect the environment of only the current process and any child processes that we invoke. We cannot affect the environment of the parent process, which is often a shell. Nevertheless, it is still useful to be able to modify the environment list.) Unfortunately, not all systems support this capability. Figure 7.8 shows the functions that are supported by the various standards and implementations.

Function	ISO C	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

Figure 7.8 Support for various environment list functions

The `clearenv` function is not part of the Single UNIX Specification. It is used to remove all entries from the environment list.

The prototypes for the middle three functions listed in Figure 7.8 are

```
#include <stdlib.h>

int putenv(char *str);
```

Returns: 0 if OK, nonzero on error

```
int setenv(const char *name, const char *value, int rewrite);

int unsetenv(const char *name);
```

Both return: 0 if OK, –1 on error

The operation of these three functions is as follows:

- The `putenv` function takes a string of the form *name=value* and places it in the environment list. If *name* already exists, its old definition is first removed.
- The `setenv` function sets *name* to *value*. If *name* already exists in the environment, then (a) if *rewrite* is nonzero, the existing definition for *name* is first removed; or (b) if *rewrite* is 0, an existing definition for *name* is not removed, *name* is not set to the new *value*, and no error occurs.
- The `unsetenv` function removes any definition of *name*. It is not an error if such a definition does not exist.

Note the difference between `putenv` and `setenv`. Whereas `setenv` must allocate memory to create the *name=value* string from its arguments, `putenv` is free to place the string passed to it directly into the environment. Indeed, many implementations do exactly this, so it would be an error to pass `putenv` a string allocated on the stack, since the memory would be reused after we return from the current function.

It is interesting to examine how these functions must operate when modifying the environment list. Recall Figure 7.6: the environment list—the array of pointers to the actual *name=value* strings—and the environment strings are typically stored at the top of a process's memory space, above the stack. Deleting a string is simple; we just find the pointer in the environment list and move all subsequent pointers down one. But adding a string or modifying an existing string is more difficult. The space at the top of the stack cannot be expanded, because it is often at the top of the address space of the

process and so can't expand upward; it can't be expanded downward, because all the stack frames below it can't be moved.

1. If we're modifying an existing *name:*
   a. If the size of the new *value* is less than or equal to the size of the existing *value*, we can just copy the new string over the old string.
   b. If the size of the new *value* is larger than the old one, however, we must `malloc` to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for *name* with the pointer to this allocated area.
2. If we're adding a new *name*, it's more complicated. First, we have to call `malloc` to allocate room for the *name=value* string and copy the string to this area.
   a. Then, if it's the first time we've added a new *name*, we have to call `malloc` to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the *name=value* string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set `environ` to point to this new list of pointers. Note from Figure 7.6 that if the original environment list was contained above the top of the stack, as is common, then we have moved this list of pointers to the heap. But most of the pointers in this list still point to *name=value* strings above the top of the stack.
   b. If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call `realloc` to allocate room for one more pointer. The pointer to the new *name=value* string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

7.10 `setjmp` and `longjmp` Functions

In C, we can't `goto` a label that's in another function. Instead, we must use the `setjmp` and `longjmp` functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

Consider the skeleton in Figure 7.9. It consists of a main loop that reads lines from standard input and calls the function `do_line` to process each line. This function then calls `get_token` to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a `switch` statement selects each command. For the single command shown, the function `cmd_add` is called.

The skeleton in Figure 7.9 is typical for programs that read commands, determine the command type, and then call functions to process each command. Figure 7.10 shows what the stack could look like after `cmd_add` has been called.

```
#include "apue.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)      /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
                break;
        }
    }
}

void
cmd_add(void)
{
    int     token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

Figure 7.9 Typical program skeleton for command processing

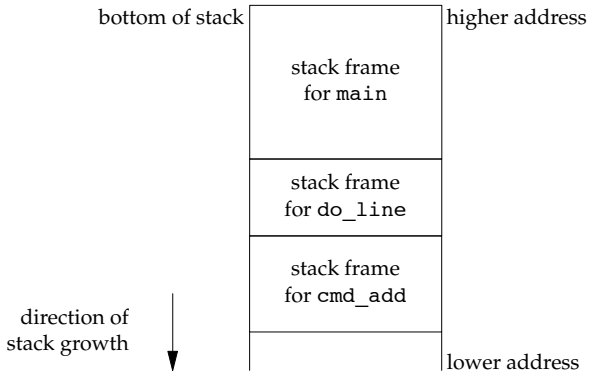


Figure 7.10 Stack frames after `cmd_add` has been called

Storage for the automatic variables is within the stack frame for each function. The array `line` is in the stack frame for `main`, the integer `cmd` is in the stack frame for `do_line`, and the integer `token` is in the stack frame for `cmd_add`.

As we've said, this type of arrangement of the stack is typical, but not required. Stacks do not have to grow toward lower memory addresses. On systems that don't have built-in hardware support for stacks, a C implementation might use a linked list for its stack frames.

The coding problem that's often encountered with programs like the one shown in Figure 7.9 is how to handle nonfatal errors. For example, if the `cmd_add` function encounters an error—say, an invalid number—it might want to print an error message, ignore the rest of the input line, and return to the `main` function to read the next input line. But when we're deeply nested numerous levels down from the `main` function, this is difficult to do in C. (In this example, the `cmd_add` function is only two levels down from `main`, but it's not uncommon to be five or more levels down from the point to which we want to return.) It becomes messy if we have to code each function with a special return value that tells it to return one level.

The solution to this problem is to use a nonlocal `goto`: the `setjmp` and `longjmp` functions. The adjective "nonlocal" indicates that we're not doing a normal C `goto` statement within a function; instead, we're branching back through the call frames to a function that is in the call path of the current function.

```
#include <setjmp.h>

int setjmp(jmp_buf env);

                Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val);
```

We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument *env* is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the *env* variable is a global variable, since we'll need to reference it from another function.

When we encounter an error—say, in the cmd_add function—we call longjmp with two arguments. The first is the same *env* that we used in a call to setjmp, and the second, *val*, is a nonzero value that becomes the return value from setjmp. The second argument allows us to use more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a *val* of 1 and also call longjmp from get_token with a *val* of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.

Let's return to the example. Figure 7.11 shows both the main and cmd_add functions. (The other two functions, do_line and get_token, haven't changed.)

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

 . . .

void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)      /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

Figure 7.11 Example of setjmp and longjmp

When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected. Before the call to longjmp in cmd_add, the stack looks like that in Figure 7.10. But longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line (Figure 7.12). Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

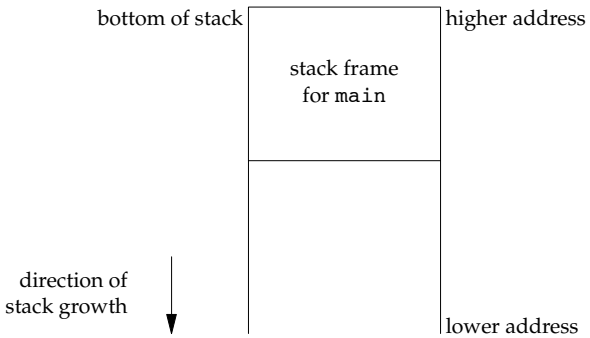


Figure 7.12 Stack frame after longjmp has been called

Automatic, Register, and Volatile Variables

We've seen what the stack looks like after calling longjmp. The next question is, "What are the states of the automatic variables and register variables in the main function?" When we return to main as a result of the longjmp, do these variables have values corresponding to those when the setjmp was previously called (i.e., are their values rolled back), or are their values left alone so that their values are whatever they were when do_line was called (which caused cmd_add to be called, which caused longjmp to be called)? Unfortunately, the answer is "It depends." Most implementations do not try to roll back these automatic variables and register variables, but the standards say only that their values are indeterminate. If you have an automatic variable that you don't want rolled back, define it with the volatile attribute. Variables that are declared as global or static are left alone when longjmp is executed.

Example

The program in Figure 7.13 demonstrates the different behavior that can be seen with automatic, global, register, static, and volatile variables after calling longjmp.

```
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf  jmpbuffer;
static int      globval;

int
main(void)
{
    int             autoval;
    register int    regival;
    volatile int    volaval;
    static int      statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

Figure 7.13 Effect of `longjmp` on various types of variables

If we compile and test the program in Figure 7.13, with and without compiler optimizations, the results are different:

```
$ gcc testjmp.c                compile without any optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ gcc -O testjmp.c             compile with full optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

Note that the optimizations don't affect the global, static, and volatile variables; their values after the `longjmp` are the last values that they assumed. The `setjmp`(3) manual page on one system states that variables stored in memory will have values as of the time of the `longjmp`, whereas variables in the CPU and floating-point registers are restored to their values when `setjmp` was called. This is indeed what we see when we run the program in Figure 7.13. Without optimization, all five variables are stored in memory (the `register` hint is ignored for `regival`). When we enable optimization, both `autoval` and `regival` go into registers, even though the former wasn't declared `register`, and the `volatile` variable stays in memory. The important thing to realize with this example is that you must use the `volatile` attribute if you're writing portable code that uses nonlocal jumps. Anything else can change from one system to the next.

Some `printf` format strings in Figure 7.13 are longer than will fit comfortably for display in a programming text. Instead of making multiple calls to `printf`, we rely on ISO C's string concatenation feature, where the sequence

```
"string1" "string2"
```

is equivalent to

```
"string1string2"
```

□

We'll return to these two functions, `setjmp` and `longjmp`, in Chapter 10 when we discuss signal handlers and their signal versions: `sigsetjmp` and `siglongjmp`.

Potential Problem with Automatic Variables

Having looked at the way stack frames are usually handled, it is worth looking at a potential error in dealing with automatic variables. The basic rule is that an automatic variable can never be referenced after the function that declared it returns. Numerous warnings about this can be found throughout the UNIX System manuals.

Figure 7.14 shows a function called `open_data` that opens a standard I/O stream and sets the buffering for the stream.

```
#include    <stdio.h>

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];  /* setvbuf makes this the stdio buffer */

    if ((fp = fopen("datafile", "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp);     /* error */
}
```

Figure 7.14 Incorrect usage of an automatic variable

The problem is that when `open_data` returns, the space it used on the stack will be used by the stack frame for the next function that is called. But the standard I/O library will still be using that portion of memory for its stream buffer. Chaos is sure to result. To correct this problem, the array `databuf` needs to be allocated from global memory, either statically (`static` or `extern`) or dynamically (one of the `alloc` functions).

7.11 `getrlimit` and `setrlimit` Functions

Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions.

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

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, –1 on error

These two functions are defined in the XSI option in the Single UNIX Specification. The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.

Each call to these two functions specifies a single *resource* and a pointer to the following structure:

```
struct rlimit {
  rlim_t  rlim_cur;  /* soft limit: current limit */
  rlim_t  rlim_max;  /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant `RLIM_INFINITY`.

The *resource* argument takes on one of the following values. Figure 7.15 shows which limits are defined by the Single UNIX Specification and supported by each implementation.

`RLIMIT_AS`	The maximum size in bytes of a process's total available memory. This affects the `sbrk` function (Section 1.11) and the `mmap` function (Section 14.8).
`RLIMIT_CORE`	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
`RLIMIT_CPU`	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the `SIGXCPU` signal is sent to the process.
`RLIMIT_DATA`	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap from Figure 7.6.
`RLIMIT_FSIZE`	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the `SIGXFSZ` signal.
`RLIMIT_MEMLOCK`	The maximum amount of memory in bytes that a process can lock into memory using `mlock`(2).
`RLIMIT_MSGQUEUE`	The maximum amount of memory in bytes that a process can allocate for POSIX message queues.
`RLIMIT_NICE`	The limit to which a process's nice value (Section 8.16) can be raised to affect its scheduling priority.
`RLIMIT_NOFILE`	The maximum number of open files per process. Changing this limit affects the value returned by the `sysconf` function for its `_SC_OPEN_MAX` argument (Section 2.5.4). See Figure 2.17 also.
`RLIMIT_NPROC`	The maximum number of child processes per real user ID. Changing this limit affects the value returned for `_SC_CHILD_MAX` by the `sysconf` function (Section 2.5.4).
`RLIMIT_NPTS`	The maximum number of pseudo terminals (Chapter 19) that a user can have open at one time.

RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time.
RLIMIT_SIGPENDING	The maximum number of signals that can be queued for a process. This limit is enforced by the sigqueue function (Section 10.20).
RLIMIT_STACK	The maximum size in bytes of the stack. See Figure 7.6.
RLIMIT_SWAP	The maximum amount of swap space in bytes that a user can consume.
RLIMIT_VMEM	This is a synonym for RLIMIT_AS.

Limit	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
RLIMIT_AS	•	•	•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_MSGQUEUE			•		
RLIMIT_NICE			•		
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_NPTS		•			
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_SIGPENDING			•		
RLIMIT_STACK	•	•	•	•	•
RLIMIT_SWAP		•			
RLIMIT_VMEM					•

Figure 7.15 Support for resource limits

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes. Indeed, the Bourne shell, the GNU Bourne-again shell, and the Korn shell have the built-in `ulimit` command, and the C shell has the built-in `limit` command. (The `umask` and `chdir` functions also have to be handled as shell built-ins.)

Example

The program in Figure 7.16 prints out the current soft limit and hard limit for all the resource limits supported on the system. To compile this program on all the various

implementations, we have conditionally included the resource names that differ. Note that some systems define `rlim_t` to be an `unsigned long long` instead of an `unsigned long`. This definition can even change on the same system, depending on whether we compile the program to support 64-bit files. Some limits apply to file size, so the `rlim_t` type has to be large enough to represent a file size limit. To avoid compiler warnings that use the wrong format specification, we first copy the limit into a 64-bit integer so that we have to deal with only one format.

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

#define doit(name)  pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
#ifdef  RLIMIT_AS
    doit(RLIMIT_AS);
#endif

    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);

#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif

#ifdef RLIMIT_MSGQUEUE
    doit(RLIMIT_MSGQUEUE);
#endif

#ifdef RLIMIT_NICE
    doit(RLIMIT_NICE);
#endif

    doit(RLIMIT_NOFILE);

#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif

#ifdef RLIMIT_NPTS
    doit(RLIMIT_NPTS);
#endif

#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif

#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
```

```
#endif

#ifdef RLIMIT_SIGPENDING
    doit(RLIMIT_SIGPENDING);
#endif

    doit(RLIMIT_STACK);

#ifdef RLIMIT_SWAP
    doit(RLIMIT_SWAP);
#endif

#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif

    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit       limit;
    unsigned long long  lim;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY) {
        printf("(infinite)  ");
    } else {
        lim = limit.rlim_cur;
        printf("%10lld  ", lim);
    }
    if (limit.rlim_max == RLIM_INFINITY) {
        printf("(infinite)");
    } else {
        lim = limit.rlim_max;
        printf("%10lld", lim);
    }
    putchar((int)'\n');
}
```

Figure 7.16 Print the current resource limits

Note that we've used the ISO C string-creation operator (#) in the `doit` macro, to generate the string value for each resource name. When we say

```
doit(RLIMIT_CORE);
```

the C preprocessor expands this into

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

Running this program under FreeBSD gives us the following output:

```
$ ./a.out
RLIMIT_AS       (infinite)  (infinite)
RLIMIT_CORE     (infinite)  (infinite)
RLIMIT_CPU      (infinite)  (infinite)
RLIMIT_DATA      536870912   536870912
RLIMIT_FSIZE    (infinite)  (infinite)
RLIMIT_MEMLOCK  (infinite)  (infinite)
RLIMIT_NOFILE         3520        3520
RLIMIT_NPROC          1760        1760
RLIMIT_NPTS     (infinite)  (infinite)
RLIMIT_RSS      (infinite)  (infinite)
RLIMIT_SBSIZE   (infinite)  (infinite)
RLIMIT_STACK      67108864    67108864
RLIMIT_SWAP     (infinite)  (infinite)
RLIMIT_VMEM     (infinite)  (infinite)
```

Solaris gives us the following results:

```
$ ./a.out
RLIMIT_AS       (infinite)  (infinite)
RLIMIT_CORE     (infinite)  (infinite)
RLIMIT_CPU      (infinite)  (infinite)
RLIMIT_DATA     (infinite)  (infinite)
RLIMIT_FSIZE    (infinite)  (infinite)
RLIMIT_NOFILE          256       65536
RLIMIT_STACK       8388608  (infinite)
RLIMIT_VMEM     (infinite)  (infinite)
```

□

Exercise 10.11 continues the discussion of resource limits, after we've covered signals.

7.12 Summary

Understanding the environment of a C program within a UNIX system's environment is a prerequisite to understanding the process control features of the UNIX System. In this chapter, we've looked at how a process is started, how it can terminate, and how it's passed an argument list and an environment. Although both the argument list and the environment are uninterpreted by the kernel, it is the kernel that passes both from the caller of `exec` to the new process.

We've also examined the typical memory layout of a C program and seen how a process can dynamically allocate and free memory. It is worthwhile to look in detail at the functions available for manipulating the environment, since they involve memory allocation. The functions `setjmp` and `longjmp` were presented, providing a way to perform nonlocal branching within a process. We finished the chapter by describing the resource limits that various implementations provide.

Exercises

7.1 On an Intel x86 system under Linux, if we execute the program that prints "hello, world" and do not call `exit` or `return`, the termination status of the program—which we can examine with the shell—is 13. Why?

7.2 When is the output from the `printfs` in Figure 7.3 actually output?

7.3 Is there any way for a function that is called by `main` to examine the command-line arguments without (a) passing `argc` and `argv` as arguments from `main` to the function or (b) having `main` copy `argc` and `argv` into global variables?

7.4 Some UNIX system implementations purposely arrange that, when a program is executed, location 0 in the data segment is not accessible. Why?

7.5 Use the `typedef` facility of C to define a new data type `Exitfunc` for an exit handler. Redo the prototype for `atexit` using this data type.

7.6 If we allocate an array of `long`s using `calloc`, is the array initialized to 0? If we allocate an array of pointers using `calloc`, is the array initialized to null pointers?

7.7 In the output from the `size` command at the end of Section 7.6, why aren't any sizes given for the heap and the stack?

7.8 In Section 7.7, the two file sizes (879443 and 8378) don't equal the sums of their respective text and data sizes. Why?

7.9 In Section 7.7, why does the size of the executable file differ so dramatically when we use shared libraries for such a trivial program?

7.10 At the end of Section 7.10, we showed how a function can't return a pointer to an automatic variable. Is the following code correct?

```
int
f1(int val)
{
    int     num = 0;
    int     *ptr = &num;

    if (val == 0) {
        int     val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```

8

Process Control

8.1 Introduction

We now turn to the process control provided by the UNIX System. This includes the creation of new processes, program execution, and process termination. We also look at the various IDs that are the property of the process—real, effective, and saved; user and group IDs—and how they're affected by the process control primitives. Interpreter files and the `system` function are also covered. We conclude the chapter by looking at the process accounting provided by most UNIX systems. This lets us look at the process control functions from a different perspective.

8.2 Process Identifiers

Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness. For example, applications sometimes include the process ID as part of a filename in an attempt to generate unique filenames.

Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

There are some special processes, but the details differ from implementation to implementation. Process ID 0 is usually the scheduler process and is often known as the *swapper*. No program on disk corresponds to this process, which is part of the

kernel and is known as a system process. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was `/etc/init` in older versions of the UNIX System and is `/sbin/init` in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. `init` usually reads the system-dependent initialization files—the `/etc/rc*` files or `/etc/inittab` and the files in `/etc/init.d`—and brings the system to a certain state, such as multiuser. The `init` process never dies. It is a normal user process, not a system process within the kernel, like the swapper, although it does run with superuser privileges. Later in this chapter, we'll see how `init` becomes the parent process of any orphaned child process.

> In Mac OS X 10.4, the `init` process was replaced with the `launchd` process, which performs the same set of tasks as `init`, but has expanded functionality. See Section 5.10 in Singh [2006] for a discussion of how `launchd` operates.

Each UNIX System implementation has its own set of kernel processes that provide operating system services. For example, on some virtual memory implementations of the UNIX System, process ID 2 is the *pagedaemon*. This process is responsible for supporting the paging of the virtual memory system.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

```
#include <unistd.h>

pid_t getpid(void);
                                        Returns: process ID of calling process

pid_t getppid(void);
                                    Returns: parent process ID of calling process

uid_t getuid(void);
                                        Returns: real user ID of calling process

uid_t geteuid(void);
                                    Returns: effective user ID of calling process

gid_t getgid(void);
                                        Returns: real group ID of calling process

gid_t getegid(void);
                                    Returns: effective group ID of calling process
```

Note that none of these functions has an error return. We'll return to the parent process ID in the next section when we discuss the `fork` function. The real and effective user and group IDs were discussed in Section 4.4.

8.3 fork Function

An existing process can create a new one by calling the fork function.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

The new process created by fork is called the *child process*. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child do share the text segment, however (Section 7.6).

Modern implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called *copy-on-write* (COW) is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system. Section 9.2 of Bach [1986] and Sections 5.6 and 5.7 of McKusick et al. [1996] provide more detail on this feature.

Variations of the fork function are provided by some platforms. All four platforms discussed in this book support the vfork(2) variant discussed in the next section.

Linux 3.2.0 also provides new process creation through the clone(2) system call. This is a generalized form of fork that allows the caller to control what is shared between parent and child.

FreeBSD 8.0 provides the rfork(2) system call, which is similar to the Linux clone system call. The rfork call is derived from the Plan 9 operating system (Pike et al. [1995]).

Solaris 10 provides two threads libraries: one for POSIX threads (pthreads) and one for Solaris threads. In previous releases, the behavior of fork differed between the two thread libraries. For POSIX threads, fork created a process containing only the calling thread, but for Solaris threads, fork created a process containing copies of all threads from the process of the calling thread. In Solaris 10, this behavior has changed; fork creates a child containing a copy of the calling thread only, regardless of which thread library is used. Solaris also provides the fork1 function, which can be used to create a process that duplicates only the calling thread, and the forkall function, which can be used to create a process that duplicates all the threads in the process. Threads are discussed in detail in Chapters 11 and 12.

Example

The program in Figure 8.1 demonstrates the `fork` function, showing how changes to variables in a child process do not affect the value of the variables in the parent process.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.1 Example of `fork` function

If we execute this program, we get

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89      child's variables were changed
pid = 429, glob = 6, var = 88      parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

In general, we never know whether the child starts executing before the parent, or vice versa. The order depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize their actions, some form of interprocess

communication is required. In the program shown in Figure 8.1, we simply have the parent put itself to sleep for 2 seconds, to let the child execute. There is no guarantee that the length of this delay is adequate, and we talk about this and other types of synchronization in Section 8.9 when we discuss race conditions. In Section 10.16, we show how to use signals to synchronize a parent and a child after a `fork`.

When we write to standard output, we subtract 1 from the size of `buf` to avoid writing the terminating null byte. Although `strlen` will calculate the length of a string not including the terminating null byte, `sizeof` calculates the size of the buffer, which does include the terminating null byte. Another difference is that using `strlen` requires a function call, whereas `sizeof` calculates the buffer length at compile time, as the buffer is initialized with a known string and its size is fixed.

Note the interaction of `fork` with the I/O functions in the program in Figure 8.1. Recall from Chapter 3 that the `write` function is not buffered. Because `write` is called before the `fork`, its data is written once to standard output. The standard I/O library, however, is buffered. Recall from Section 5.12 that standard output is line buffered if it's connected to a terminal device; otherwise, it's fully buffered. When we run the program interactively, we get only a single copy of the first `printf` line, because the standard output buffer is flushed by the newline. When we redirect standard output to a file, however, we get two copies of the `printf` line. In this second case, the `printf` before the `fork` is called once, but the line remains in the buffer when `fork` is called. This buffer is then copied into the child when the parent's data space is copied to the child. Both the parent and the child now have a standard I/O buffer with this line in it. The second `printf`, right before the `exit`, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed. □

File Sharing

When we redirect the standard output of the parent from the program in Figure 8.1, the child's standard output is also redirected. Indeed, one characteristic of `fork` is that all file descriptors that are open in the parent are duplicated in the child. We say "duplicated" because it's as if the `dup` function had been called for each descriptor. The parent and the child share a file table entry for every open descriptor (recall Figure 3.9).

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from `fork`, we have the arrangement shown in Figure 8.2.

It is important that the parent and the child share the same file offset. Consider a process that `fork`s a child, then `wait`s for the child to complete. Assume that both processes write to standard output as part of their normal processing. If the parent has its standard output redirected (by a shell, perhaps), it is essential that the parent's file offset be updated by the child when the child writes to standard output. In this case, the child can write to standard output while the parent is `wait`ing for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote. If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

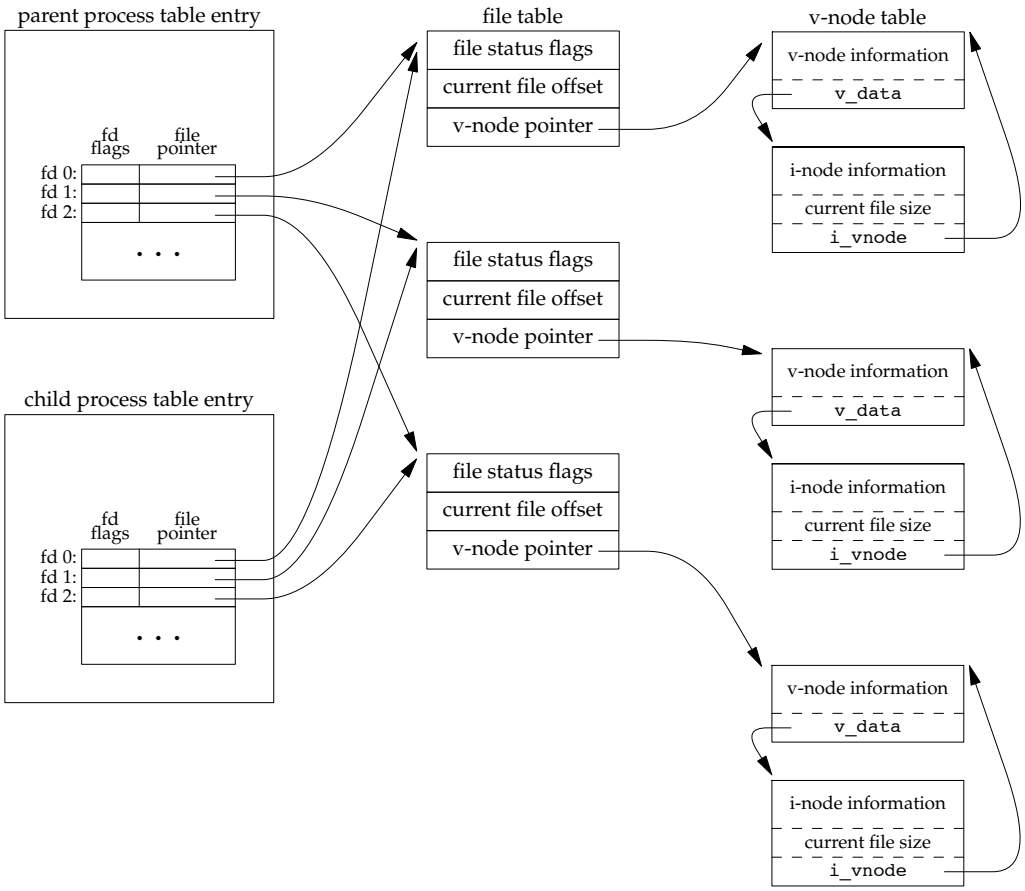


Figure 8.2 Sharing of open files between parent and child after `fork`

If both parent and child write to the same descriptor, without any form of synchronization, such as having the parent `wait` for the child, their output will be intermixed (assuming it's a descriptor that was open before the `fork`). Although this is possible—we saw it in Figure 8.2—it's not the normal mode of operation.

There are two normal cases for handling the descriptors after a `fork`.

1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
2. Both the parent and the child go their own ways. Here, after the `fork`, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often found with network servers.

Besides the open files, numerous other properties of the parent are inherited by the child:

- Real user ID, real group ID, effective user ID, and effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- The return values from `fork` are different.
- The process IDs are different.
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change.
- The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0 (these times are discussed in Section 8.17).
- File locks set by the parent are not inherited by the child.
- Pending alarms are cleared for the child.
- The set of pending signals for the child is set to the empty set.

Many of these features haven't been discussed yet—we'll cover them in later chapters.

The two main reasons for `fork` to fail are (a) if too many processes are already in the system, which usually means that something else is wrong, or (b) if the total number of processes for this real user ID exceeds the system's limit. Recall from Figure 2.11 that `CHILD_MAX` specifies the maximum number of simultaneous processes per real user ID.

There are two uses for `fork`:

1. When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time. This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls `fork` and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case, the child does an `exec` (which we describe in Section 8.10) right after it returns from the `fork`.

Some operating systems combine the operations from step 2—a `fork` followed by an `exec`—into a single operation called a *spawn*. The UNIX System separates the two, as there are numerous cases where it is useful to `fork` without doing an `exec`. Also, separating the two operations allows the child to change the per-process attributes between the `fork` and the `exec`, such as I/O redirection, user ID, signal disposition, and so on. We'll see numerous examples of this in Chapter 15.

> The Single UNIX Specification does include `spawn` interfaces in the advanced real-time option group. These interfaces are not intended to be replacements for `fork` and `exec`, however. They are intended to support systems that have difficulty implementing `fork` efficiently, especially systems without hardware support for memory management.

8.4 `vfork` Function

The function `vfork` has the same calling sequence and same return values as `fork`, but the semantics of the two functions differ.

> The `vfork` function originated with 2.9BSD. Some consider the function a blemish, but all the platforms covered in this book support it. In fact, the BSD developers removed it from the 4.4BSD release, but all the open source BSD distributions that derive from 4.4BSD added support for it back into their own releases. The `vfork` function was marked as an obsolescent interface in Version 3 of the Single UNIX Specification and was removed entirely in Version 4. We include it here for historical reasons only. Portable applications should not use it.

The `vfork` function was intended to create a new process for the purpose of executing a new program (step 2 at the end of the previous section), similar to the method used by the bare-bones shell from Figure 1.7. The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls `exec` (or `exit`) right after the `vfork`. Instead, the child runs in the address space of the parent until it calls either `exec` or `exit`. This optimization is more efficient on some implementations of the UNIX System, but leads to undefined results if the child modifies any data (except the variable used to hold the return value from `vfork`), makes function calls, or returns without calling `exec` or `exit`. (As we mentioned in the previous section, implementations use copy-on-write to improve the efficiency of a `fork` followed by an `exec`, but no copying is still faster than some copying.)

Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

Example

The program in Figure 8.3 is a modified version of the program from Figure 8.1. We've replaced the call to `fork` with `vfork` and removed the `write` to standard output. Also, we don't need to have the parent call `sleep`, as we're guaranteed that it is put to sleep by the kernel until the child calls either `exec` or `exit`.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /* parent continues here */
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.3 Example of `vfork` function

Running this program gives us

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

Here, the incrementing of the variables done by the child changes the values in the parent. Because the child runs in the address space of the parent, this doesn't surprise us. This behavior, however, differs from the behavior of `fork`.

Note in Figure 8.3 that we call `_exit` instead of `exit`. As we described in Section 7.3, `_exit` does not perform any flushing of standard I/O buffers. If we call `exit` instead, the results are indeterminate. Depending on the implementation of the standard I/O library, we might see no difference in the output, or we might find that the output from the first `printf` in the parent has disappeared.

If the child calls `exit`, the implementation flushes the standard I/O streams. If this is the only action taken by the library, then we will see no difference from the output generated if the child called `_exit`. If the implementation also closes the standard I/O streams, however, the memory representing the `FILE` object for the standard output will be cleared out. Because the child is borrowing the parent's address space, when the parent resumes and calls `printf`, no output will appear and `printf` will return –1. Note that the parent's `STDOUT_FILENO` is still valid, as the child gets a copy of the parent's file descriptor array (refer back to Figure 8.2).

Most modern implementations of `exit` do not bother to close the streams. Because the process is about to exit, the kernel will close all the file descriptors open in the process. Closing them in the library simply adds overhead without any benefit.

□

Section 5.6 of McKusick et al. [1996] contains additional information on the implementation issues of `fork` and `vfork`. Exercises 8.1 and 8.2 continue the discussion of `vfork`.

8.5 `exit` Functions

As we described in Section 7.3, a process can terminate normally in five ways:

1. Executing a `return` from the `main` function. As we saw in Section 7.3, this is equivalent to calling `exit`.
2. Calling the `exit` function. This function is defined by ISO C and includes the calling of all exit handlers that have been registered by calling `atexit` and closing all standard I/O streams. Because ISO C does not deal with file descriptors, multiple processes (parents and children), and job control, the definition of this function is incomplete for a UNIX system.
3. Calling the `_exit` or `_Exit` function. ISO C defines `_Exit` to provide a way for a process to terminate without running exit handlers or signal handlers. Whether standard I/O streams are flushed depends on the implementation. On UNIX systems, `_Exit` and `_exit` are synonymous and do not flush standard I/O streams. The `_exit` function is called by `exit` and handles the UNIX system-specific details; `_exit` is specified by POSIX.1.

   In most UNIX system implementations, `exit`(3) is a function in the standard C library, whereas `_exit`(2) is a system call.

4. Executing a `return` from the start routine of the last thread in the process. The return value of the thread is not used as the return value of the process, however. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the `pthread_exit` function from the last thread in the process. As with the previous case, the exit status of the process in this situation is always 0, regardless of the argument passed to `pthread_exit`. We'll say more about `pthread_exit` in Section 11.5.

The three forms of abnormal termination are as follows:

1. Calling `abort`. This is a special case of the next item, as it generates the `SIGABRT` signal.
2. When the process receives certain signals. (We describe signals in more detail in Chapter 10.) The signal can be generated by the process itself (e.g., by calling the `abort` function), by some other process, or by the kernel. Examples of

signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.

3. The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later the target thread terminates. We discuss cancellation requests in detail in Sections 11.5 and 12.7.

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and so on.

For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated. For the three exit functions (`exit`, `_exit`, and `_Exit`), this is done by passing an exit status as the argument to the function. In the case of an abnormal termination, however, the kernel—not the process—generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the `wait` or the `waitpid` function (described in the next section).

Note that we differentiate between the exit status, which is the argument to one of the three exit functions or the return value from `main`, and the termination status. The exit status is converted into a termination status by the kernel when `_exit` is finally called (recall Figure 7.2). Figure 8.4 describes the various ways the parent can examine the termination status of a child. If the child terminated normally, the parent can obtain the exit status of the child.

When we described the `fork` function, it was obvious that the child has a parent process after the call to `fork`. Now we're talking about returning a termination status to the parent. But what happens if the parent terminates before the child? The answer is that the `init` process becomes the parent process of any process whose parent terminates. In such a case, we say that the process has been inherited by `init`. What normally happens is that whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists. If so, the parent process ID of the surviving process is changed to be 1 (the process ID of `init`). This way, we're guaranteed that every process has a parent.

Another condition we have to worry about is when a child terminates before its parent. If the child completely disappeared, the parent wouldn't be able to fetch its termination status when and if the parent was finally ready to check if the child had terminated. The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls `wait` or `waitpid`. Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process. The kernel can discard all the memory used by the process and close its open files. In UNIX System terminology, a process that has terminated, but whose parent has not yet waited for it, is called a *zombie*. The `ps`(1) command prints the state of a zombie process as *Z*. If we write a long-running program that `fork`s many child processes, they become zombies unless we wait for them and fetch their termination status.

Some systems provide ways to prevent the creation of zombies, as we describe in Section 10.7.

The final condition to consider is this: What happens when a process that has been inherited by `init` terminates? Does it become a zombie? The answer is "no," because `init` is written so that whenever one of its children terminates, `init` calls one of the `wait` functions to fetch the termination status. By doing this, `init` prevents the system from being clogged by zombies. When we say "one of `init`'s children," we mean either a process that `init` generates directly (such as `getty`, which we describe in Section 9.2) or a process whose parent has terminated and has been subsequently inherited by `init`.

8.6 `wait` and `waitpid` Functions

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent. Because the termination of a child is an asynchronous event—it can happen at any time while the parent is running—this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored. We describe these options in Chapter 10. For now, we need to be aware that a process that calls `wait` or `waitpid` can

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes

If the process is calling `wait` because it received the `SIGCHLD` signal, we expect `wait` to return immediately. But if we call it at any random point in time, it can block.

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

pid_t wait(int *statloc);

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

Both return: process ID if OK, 0 (see later), or –1 on error

The differences between these two functions are as follows:

- The `wait` function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, `wait` returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, `wait` returns when one terminates. We can always tell which child terminated, because the process ID is returned by the function.

For both functions, the argument *statloc* is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument. If we don't care about the termination status, we simply pass a null pointer as this argument.

Traditionally, the integer status that these two functions return has been defined by the implementation, with certain bits indicating the exit status (for a normal return), other bits indicating the signal number (for an abnormal return), one bit indicating whether a core file was generated, and so on. POSIX.1 specifies that the termination status is to be looked at using various macros that are defined in `<sys/wait.h>`. Four mutually exclusive macros tell us how the process terminated, and they all begin with `WIF`. Based on which of these four macros is true, other macros are used to obtain the exit status, signal number, and the like. The four mutually exclusive macros are shown in Figure 8.4.

Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

We'll discuss how a process can be stopped in Section 9.8 when we discuss job control.

Example

The function `pr_exit` in Figure 8.5 uses the macros from Figure 8.4 to print a description of the termination status. We'll call this function from numerous programs in the text. Note that this function handles the `WCOREDUMP` macro, if it is defined.

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

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

Figure 8.5 Print a description of the `exit` status

FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 all support the `WCOREDUMP` macro. However, some platforms hide its definition if the `_POSIX_C_SOURCE` constant is defined (recall Section 2.7).

The program shown in Figure 8.6 calls the `pr_exit` function, demonstrating the various values for the termination status. If we run the program in Figure 8.6, we get

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

For now, we print the signal number from `WTERMSIG`. We can look at the `<signal.h>` header to verify that `SIGABRT` has a value of 6 and that `SIGFPE` has a value of 8. We'll see a portable way to map a signal number to a descriptive name in Section 10.22. □

As we mentioned, if we have more than one child, `wait` returns on termination of any of the children. But what if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)? In older versions of the UNIX System, we would have to call `wait` and compare the returned process ID with the one we're interested in. If the terminated process wasn't the one we wanted, we would have to save the process ID and termination status and call `wait` again. We would need to continue doing this until the desired process terminated. The next time we wanted to wait for a specific process, we would go through the list of already terminated processes to see whether we had already waited for it, and if not, call `wait`

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

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

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

Figure 8.6 Demonstrate various `exit` statuses

again. What we need is a function that waits for a specific process. This functionality (and more) is provided by the POSIX.1 `waitpid` function.

The interpretation of the *pid* argument for `waitpid` depends on its value:

pid == –1	Waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	Waits for the child whose process ID equals *pid*.
pid == 0	Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.)
pid < –1	Waits for any child whose process group ID equals the absolute value of *pid*.

The `waitpid` function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by *statloc*. With `wait`, the only real error is if the calling process has no children. (Another error return is possible, in case the function call is interrupted by a signal. We'll discuss this in Chapter 10.) With `waitpid`, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The *options* argument lets us further control the operation of `waitpid`. This argument either is 0 or is constructed from the bitwise OR of the constants in Figure 8.7.

> FreeBSD 8.0 and Solaris 10 support one additional, but nonstandard, *option* constant. `WNOWAIT` has the system keep the process whose termination status is returned by `waitpid` in a wait state, so that it may be waited for again.

Constant	Description
`WCONTINUED`	If the implementation supports job control, the status of any child specified by *pid* that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI option).
`WNOHANG`	The `waitpid` function will not block if a child specified by *pid* is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by *pid* that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

Figure 8.7 The *options* constants for `waitpid`

The `waitpid` function provides three features that aren't provided by the `wait` function.

1. The `waitpid` function lets us wait for one particular process, whereas the `wait` function returns the status of any terminated child. We'll return to this feature when we discuss the `popen` function.
2. The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
3. The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

Example

Recall our discussion in Section 8.5 about zombie processes. If we want to write a process so that it `fork`s a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call `fork` twice. The program in Figure 8.8 does this.

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

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %ld\n", (long)getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

Figure 8.8 Avoid zombie processes by calling `fork` twice

We call `sleep` in the second child to ensure that the first child terminates before printing the parent process ID. After a `fork`, either the parent or the child can continue executing; we never know which will resume execution first. If we didn't put the second child to sleep, and if it resumed execution after the `fork` before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

Executing the program in Figure 8.8 gives us

```
$ ./a.out
$ second child, parent pid = 1
```

Note that the shell prints its prompt when the original process terminates, which is before the second child prints its parent process ID. □

8.7 waitid Function

The Single UNIX Specification includes an additional function to retrieve the exit status of a process. The `waitid` function is similar to `waitpid`, but provides extra flexibility.

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, –1 on error

Like `waitpid`, `waitid` allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used. The *id* parameter is interpreted based on the value of *idtype*. The types supported are summarized in Figure 8.9.

Constant	Description
`P_PID`	Wait for a particular process: *id* contains the process ID of the child to wait for.
`P_PGID`	Wait for any child process in a particular process group: *id* contains the process group ID of the children to wait for.
`P_ALL`	Wait for any child process: *id* is ignored.

Figure 8.9 The *idtype* constants for `waitid`

The *options* argument is a bitwise OR of the flags shown in Figure 8.10. These flags indicate which state changes the caller is interested in.

Constant	Description
`WCONTINUED`	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
`WEXITED`	Wait for processes that have exited.
`WNOHANG`	Return immediately instead of blocking if there is no child exit status available.
`WNOWAIT`	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to `wait`, `waitid`, or `waitpid`.
`WSTOPPED`	Wait for a process that has stopped and whose status has not yet been reported.

Figure 8.10 The *options* constants for `waitid`

At least one of `WCONTINUED`, `WEXITED`, or `WSTOPPED` must be specified in the *options* argument.

The *infop* argument is a pointer to a `siginfo` structure. This structure contains detailed information about the signal generated that caused the state change in the child process. The `siginfo` structure is discussed further in Section 10.14.

> Of the four platforms covered in this book, only Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 provide support for `waitid`. Note, however, that Mac OS X 10.6.8 doesn't set all the information we expect in the `siginfo` structure.

14

Advanced I/O

14.1 Introduction

This chapter covers numerous topics and functions that we lump under the term *advanced I/O*: nonblocking I/O, record locking, I/O multiplexing (the `select` and `poll` functions), asynchronous I/O, the `readv` and `writev` functions, and memory-mapped I/O (`mmap`). We need to cover these topics before describing interprocess communication in Chapter 15, Chapter 17, and many of the examples in later chapters.

14.2 Nonblocking I/O

In Section 10.5, we said that system calls are divided into two categories: the "slow" ones and all the others. The slow system calls are those that can block forever. They include

- Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types (e.g., no room in the pipe, network flow control)
- Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone, or an open of a FIFO for writing only, when no other process has the FIFO open for reading)
- Reads and writes of files that have mandatory record locking enabled

- Certain `ioctl` operations
- Some of the interprocess communication functions (Chapter 15)

We also said that system calls related to disk I/O are not considered slow, even though the read or write of a disk file can block the caller temporarily.

Nonblocking I/O lets us issue an I/O operation, such as an `open`, `read`, or `write`, and not have it block forever. If the operation cannot be completed, the call returns immediately with an error noting that the operation would have blocked.

There are two ways to specify nonblocking I/O for a given descriptor.

1. If we call `open` to get the descriptor, we can specify the `O_NONBLOCK` flag (Section 3.3).
2. For a descriptor that is already open, we call `fcntl` to turn on the `O_NONBLOCK` file status flag (Section 3.14). Figure 3.12 shows a function that we can call to turn on any of the file status flags for a descriptor.

> Earlier versions of System V used the flag `O_NDELAY` to specify nonblocking mode. These versions of System V returned a value of 0 from the `read` function if there wasn't any data to be read. Since this use of a return value of 0 overlapped with the normal UNIX System convention of 0 meaning the end of file, POSIX.1 chose to provide a nonblocking flag with a different name and different semantics. Indeed, with these older versions of System V, when we get a return of 0 from `read`, we don't know whether the call would have blocked or whether the end of file was encountered. We'll see that POSIX.1 requires that `read` return –1 with `errno` set to `EAGAIN` if there is no data to read from a nonblocking descriptor. Some platforms derived from System V support both the older `O_NDELAY` and the POSIX.1 `O_NONBLOCK`, but in this text we'll use only the POSIX.1 feature. The older `O_NDELAY` is intended for backward compatibility and should not be used in new applications.
>
> 4.3BSD provided the `FNDELAY` flag for `fcntl`, and its semantics were slightly different. Instead of affecting only the file status flags for the descriptor, the flags for either the terminal device or the socket were also changed to be nonblocking, thereby affecting all users of the terminal or socket, not just the users sharing the same file table entry (4.3BSD nonblocking I/O worked only on terminals and sockets). Also, 4.3BSD returned `EWOULDBLOCK` if an operation on a nonblocking descriptor could not complete without blocking. Today, BSD-based systems provide the POSIX.1 `O_NONBLOCK` flag and define `EWOULDBLOCK` to be the same as `EAGAIN`. These systems provide nonblocking semantics consistent with other POSIX-compatible systems: changes in file status flags affect all users of the same file table entry, but are independent of accesses to the same device through other file table entries. (Refer to Figures 3.7 and 3.9.)

Example

Let's look at an example of nonblocking I/O. The program in Figure 14.1 reads up to 500,000 bytes from the standard input and attempts to write it to the standard output. The standard output is first set to be nonblocking. The output is in a loop, with the results of each `write` being printed on the standard error. The function `clr_fl` is similar to the function `set_fl` that we showed in Figure 3.12. This new function simply clears one or more of the flag bits.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>

char    buf[500000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */

    exit(0);
}
```

Figure 14.1 Large nonblocking `write`

If the standard output is a regular file, we expect the `write` to be executed once:

```
$ ls -l /etc/services                              print file size
-rw-r--r--  1 root     677959 Jun 23  2009 /etc/services
$ ./a.out < /etc/services > temp.file              try a regular file first
read 500000 bytes
nwrite = 500000, errno = 0                         a single write
$ ls -l temp.file                                  verify size of output file
-rw-rw-r--  1 sar      500000 Apr  1 13:03 temp.file
```

But if the standard output is a terminal, we expect the `write` to return a partial count sometimes and an error at other times. This is what we see:

```
$ ./a.out < /etc/services 2>stderr.out          output to terminal
                                                lots of output to terminal ...
$ cat stderr.out
read 500000 bytes
nwrite = 999, errno = 0
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = 1001, errno = 0
nwrite = -1, errno = 35
nwrite = 1002, errno = 0
nwrite = 1004, errno = 0
nwrite = 1003, errno = 0
nwrite = 1003, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         61 of these errors
   .
   .
   .
nwrite = 1006, errno = 0
nwrite = 1004, errno = 0
nwrite = 1005, errno = 0
nwrite = 1006, errno = 0
nwrite = -1, errno = 35                         108 of these errors
   .
   .
   .
nwrite = 1006, errno = 0
nwrite = 1005, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         681 of these errors
   .
   .
   .
                                                and so on ...
nwrite = 347, errno = 0
```

On this system, the `errno` of 35 is `EAGAIN`. The amount of data accepted by the terminal driver varies from system to system. The results will also vary depending on how you are logged in to the system: on the system console, on a hard-wired terminal, on a network connection using a pseudo terminal. If you are running a windowing system on your terminal, you are also going through a pseudo terminal device. □

In this example, the program issues more than 9,000 `write` calls, even though only 500 are needed to output the data. The rest just return an error. This type of loop, called *polling*, is a waste of CPU time on a multiuser system. In Section 14.4, we'll see that I/O multiplexing with a nonblocking descriptor is a more efficient way to do this.

Sometimes, we can avoid using nonblocking I/O by designing our applications to use multiple threads (see Chapter 11). We can allow individual threads to block in I/O calls if we can continue to make progress in other threads. This can sometimes simplify the design, as we shall see in Chapter 21; at other times, however, the overhead of synchronization can add more complexity than is saved from using threads.

14.3 Record Locking

What happens when two people edit the same file at the same time? In most UNIX systems, the final state of the file corresponds to the last process that wrote the file. In some applications, however, such as a database system, a process needs to be certain that it alone is writing to a file. To provide this capability for processes that need it, commercial UNIX systems provide record locking. (In Chapter 20, we develop a database library that uses record locking.)

Record locking is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file. Under the UNIX System, "record" is a misnomer; the UNIX kernel does not have a notion of records in a file. A better term is *byte-range locking*, given that it is a range of a file (possibly the entire file) that is locked.

History

One of the criticisms of early UNIX systems was that they couldn't be used to run database systems, because they did not support locking portions of files. As UNIX systems found their way into business computing environments, various groups added support for record locking (differently, of course).

Early Berkeley releases supported only the `flock` function. This function locks only entire files, not regions of a file.

Record locking was added to System V Release 3 through the `fcntl` function. The `lockf` function was built on top of this, providing a simplified interface. These functions allowed callers to lock arbitrary byte ranges in a file, ranging from the entire file down to a single byte within the file.

POSIX.1 chose to standardize on the `fcntl` approach. Figure 14.2 shows the forms of record locking provided by various systems. Note that the Single UNIX Specification includes `lockf` in the XSI option.

System	Advisory	Mandatory	`fcntl`	`lockf`	`flock`
SUS	•		•	XSI	
FreeBSD 8.0	•		•	•	•
Linux 3.2.0	•	•	•	•	•
Mac OS X 10.6.8	•		•	•	•
Solaris 10	•	•	•	•	•

Figure 14.2 Forms of record locking supported by various UNIX systems

We describe the difference between advisory locking and mandatory locking later in this section. In this text, we describe only the POSIX.1 `fcntl` locking.

> Record locking was originally added to Version 7 in 1980 by John Bass. The system call entry into the kernel was a function named `locking`. This function provided mandatory record locking and propagated through many versions of System III. Xenix systems picked up this function, and some Intel-based System V derivatives, such as OpenServer 5, continued to support it in a Xenix-compatibility library.

`fcntl` Record Locking

Let's repeat the prototype for the `fcntl` function from Section 3.14.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
```

Returns: depends on *cmd* if OK (see following), –1 on error

For record locking, *cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW`. The third argument (which we'll call *flockptr*) is a pointer to an `flock` structure.

```
struct flock {
  short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;  /* offset in bytes, relative to l_whence */
  off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;    /* returned with F_GETLK */
};
```

This structure describes

- The type of lock desired: `F_RDLCK` (a shared read lock), `F_WRLCK` (an exclusive write lock), or `F_UNLCK` (unlocking a region)
- The starting byte offset of the region being locked or unlocked (`l_start` and `l_whence`)
- The size of the region in bytes (`l_len`)
- The ID (`l_pid`) of the process holding the lock that can block the current process (returned by `F_GETLK` only)

Numerous rules apply to the specification of the region to be locked or unlocked.

- The two elements that specify the starting offset of the region are similar to the last two arguments of the `lseek` function (Section 3.6). Indeed, the `l_whence` member is specified as `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If `l_len` is 0, it means that the lock extends to the largest possible offset of the file. This allows us to lock a region starting anywhere in the file, up through and including any data that is appended to the file. (We don't have to try to guess how many bytes might be appended to the file.)
- To lock the entire file, we set `l_start` and `l_whence` to point to the beginning of the file and specify a length (`l_len`) of 0. (There are several ways to specify the beginning of the file, but most applications specify `l_start` as 0 and `l_whence` as `SEEK_SET`.)

We previously mentioned two types of locks: a shared read lock (`l_type` of `F_RDLCK`) and an exclusive write lock (`F_WRLCK`). The basic rule is that any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. Furthermore, if there are one or more read locks on a byte, there can't be any write locks on that byte; if there is an exclusive write lock on a byte, there can't be any read locks on that byte. We show this compatibility rule in Figure 14.3.

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

Figure 14.3 Compatibility between different lock types

The compatibility rule applies to lock requests made from different processes, not to multiple lock requests made by a single process. If a process has an existing lock on a range of a file, a subsequent attempt to place a lock on the same range by the same process will replace the existing lock with the new one. Thus, if a process has a write lock on bytes 16–32 of a file and then tries to place a read lock on bytes 16–32, the request will succeed, and the write lock will be replaced by a read lock.

To obtain a read lock, the descriptor must be open for reading; to obtain a write lock, the descriptor must be open for writing.

We can now describe the three commands for the `fcntl` function.

`F_GETLK` Determine whether the lock described by *flockptr* is blocked by some other lock. If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by *flockptr*. If no lock exists that would prevent ours from being created, the structure pointed to by *flockptr* is left unchanged except for the `l_type` member, which is set to `F_UNLCK`.

`F_SETLK` Set the lock described by *flockptr*. If we are trying to obtain a read lock (`l_type` of `F_RDLCK`) or a write lock (`l_type` of `F_WRLCK`) and the compatibility rule prevents the system from giving us the lock (Figure 14.3), `fcntl` returns immediately with `errno` set to either `EACCES` or `EAGAIN`.

> Although POSIX allows an implementation to return either error code, all four implementations described in this text return `EAGAIN` if the locking request cannot be satisfied.

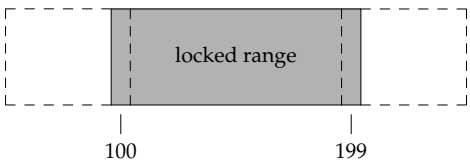
This command is also used to clear the lock described by *flockptr* (`l_type` of `F_UNLCK`).

`F_SETLKW` This command is a blocking version of `F_SETLK`. (The `W` in the command name means *wait*.) If the requested read lock or write lock cannot be granted because another process currently has some part of the requested region locked, the calling process is put to sleep. The process wakes up either when the lock becomes available or when interrupted by a signal.

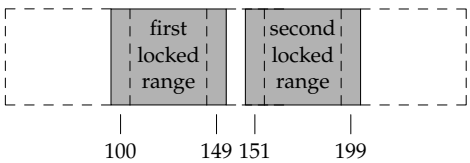
Be aware that testing for a lock with `F_GETLK` and then trying to obtain that lock with `F_SETLK` or `F_SETLKW` is not an atomic operation. We have no guarantee that, between the two `fcntl` calls, some other process won't come in and obtain the same lock. If we don't want to block while waiting for a lock to become available to us, we must handle the possible error returns from `F_SETLK`.

> Note that POSIX.1 doesn't specify what happens when one process read locks a range of a file, a second process blocks while trying to get a write lock on the same range, and a third processes then attempts to get another read lock on the range. If the third process is allowed to place a read lock on the range just because the range is already read locked, then the implementation might starve processes with pending write locks. Thus, as additional requests to read lock the same range arrive, the time that the process with the pending write-lock request has to wait is extended. If the read-lock requests arrive quickly enough without a lull in the arrival rate, then the writer could wait for a long time.

When setting or releasing a lock on a file, the system combines or splits adjacent areas as required. For example, if we lock bytes 100 through 199 and then unlock byte 150, the kernel still maintains the locks on bytes 100 through 149 and bytes 151 through 199. Figure 14.4 illustrates the byte-range locks in this situation.



File after locking bytes 100 through 199



File after unlocking byte 150

Figure 14.4 File byte-range lock diagram

If we were to lock byte 150, the system would coalesce the adjacent locked regions into a single region from byte 100 through 199. The resulting picture would be the first diagram in Figure 14.4, the same as when we started.

Example — Requesting and Releasing a Lock

To save ourselves from having to allocate an `flock` structure and fill in all the elements each time, the function `lock_reg` in Figure 14.5 handles all these details.

```
#include "apue.h"
#include <fcntl.h>

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    return(fcntl(fd, cmd, &lock));
}
```

Figure 14.5 Function to lock or unlock a region of a file

Since most locking calls are to lock or unlock a region (the command `F_GETLK` is rarely used), we normally use one of the following five macros, which are defined in `apue.h` (Appendix B).

```
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

We have purposely defined the first three arguments to these macros in the same order as the `lseek` function. □

Example — Testing for a Lock

Figure 14.6 defines the function `lock_test` that we'll use to test for a lock.

```
#include "apue.h"
#include <fcntl.h>

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
```

```
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);      /* false, region isn't locked by another proc */
    return(lock.l_pid); /* true, return pid of lock owner */
}
```

Figure 14.6 Function to test for a locking condition

If a lock exists that would block the request specified by the arguments, this function returns the process ID of the process holding the lock. Otherwise, the function returns 0 (false). We normally call this function from the following two macros (defined in `apue.h`):

```
#define is_read_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

Note that the `lock_test` function can't be used by a process to see whether it is currently holding a portion of a file locked. The definition of the `F_GETLK` command states that the information returned applies to an existing lock that would prevent us from creating our own lock. Since the `F_SETLK` and `F_SETLKW` commands always replace a process's existing lock if it exists, we can never block on our own lock; thus, the `F_GETLK` command will never report our own lock. □

Example — Deadlock

Deadlock occurs when two processes are each waiting for a resource that the other has locked. The potential for deadlock exists if a process that controls a locked region is put to sleep when it tries to lock another region that is controlled by a different process.

Figure 14.7 shows an example of deadlock. The child locks byte 0 and the parent locks byte 1. Each then tries to lock the other's already locked byte. We use the parent–child synchronization routines from Section 8.9 (`TELL_xxx` and `WAIT_xxx`) so that each process can wait for the other to obtain its lock.

```
#include "apue.h"
#include <fcntl.h>

static void
lockabyte(const char *name, int fd, off_t offset)
```

```
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);
    printf("%s: got the lock, byte %lld\n", name, (long long)offset);
}

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

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");

    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {                        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

Figure 14.7 Example of deadlock detection

Running the program in Figure 14.7 gives us

```
$ ./a.out
parent: got the lock, byte 1
child: got the lock, byte 0
parent: writew_lock error: Resource deadlock avoided
child: got the lock, byte 1
```

When a deadlock is detected, the kernel has to choose one process to receive the error return. In this example, the parent was chosen, but this is an implementation detail. On some systems, the child always receives the error. On other systems, the parent always gets the error. On some systems, you might even see the errors split between the child and the parent as multiple lock attempts are made. □

Implied Inheritance and Release of Locks

Three rules govern the automatic inheritance and release of record locks.

1. Locks are associated with a process and a file. This has two implications. The first is obvious: when a process terminates, all its locks are released. The second is far from obvious: whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released. This means that if we make the calls

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = dup(fd1);
   close(fd2);
   ```

   after the `close(fd2)`, the lock that was obtained on `fd1` is released. The same thing would happen if we replaced the `dup` with `open`, as in

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = open(pathname, ...)
   close(fd2);
   ```

   to open the same file on another descriptor.
2. Locks are never inherited by the child across a `fork`. This means that if a process obtains a lock and then calls `fork`, the child is considered another process with regard to the lock that was obtained by the parent. The child has to call `fcntl` to obtain its own locks on any descriptors that were inherited across the `fork`. This constraint makes sense because locks are meant to prevent multiple processes from writing to the same file at the same time. If the child inherited locks across a `fork`, both the parent and the child could write to the same file at the same time.
3. Locks are inherited by a new program across an `exec`. Note, however, that if the close-on-exec flag is set for a file descriptor, all locks for the underlying file are released when the descriptor is closed as part of an `exec`.

FreeBSD Implementation

Let's take a brief look at the data structures used in the FreeBSD implementation. This should help clarify rule 1, which states that locks are associated with a process and a file.

Consider a process that executes the following statements (ignoring error returns):

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);    /* parent write locks byte 0 */
if ((pid = fork()) > 0) {           /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
}
pause();
```

Figure 14.8 shows the resulting data structures after both the parent and the child have paused.

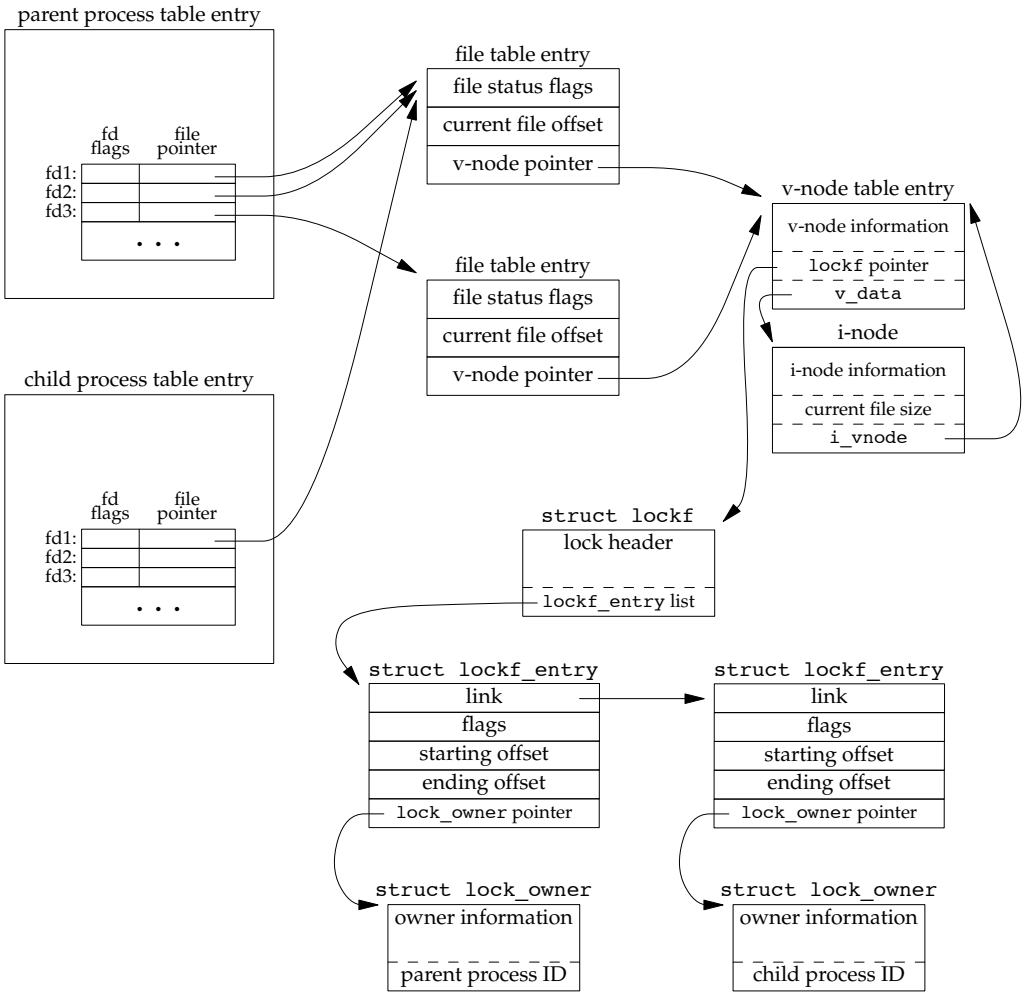


Figure 14.8 The FreeBSD data structures for record locking

We've shown the data structures that result from the `open`, `fork`, and `dup` calls earlier (Figures 3.9 and 8.2). What is new here are the `lockf` structures that are linked together from the i-node structure. Each `lockf` structure describes one locked region (defined by an offset and length) for a given process. We show two of these structures: one for the parent's call to `write_lock` and one for the child's call to `read_lock`. Each structure contains the corresponding process ID.

In the parent, closing any one of `fd1`, `fd2`, or `fd3` causes the parent's lock to be released. When any one of these three file descriptors is closed, the kernel goes through

the linked list of locks for the corresponding i-node and releases the locks held by the calling process. The kernel can't tell (and doesn't care) which descriptor of the three was used by the parent to obtain the lock.

Example

In the program in Figure 13.6, we saw how a daemon can use a lock on a file to ensure that only one copy of the daemon is running. Figure 14.9 shows the implementation of the `lockfile` function used by the daemon to place a write lock on a file.

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

int
lockfile(int fd)
{
    struct flock fl;

    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

Figure 14.9 Place a write lock on an entire file

Alternatively, we could define the `lockfile` function in terms of the `write_lock` function:

```
#define lockfile(fd) write_lock((fd), 0, SEEK_SET, 0)
```

□

Locks at End of File

We need to use caution when locking or unlocking byte ranges relative to the end of file. Most implementations convert an `l_whence` value of `SEEK_CUR` or `SEEK_END` into an absolute file offset, using `l_start` and the file's current position or current length. Often, however, we need to specify a lock relative to the file's current length, but we can't call `fstat` to obtain the current file size, since we don't have a lock on the file. (There's a chance that another process could change the file's length between the call to `fstat` and the lock call.)

Consider the following sequence of steps:

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

This sequence of code might not do what you expect. It obtains a write lock from the current end of the file onward, covering any future data we might append to the file.

Assuming that we are at end of file when we perform the first `write`, this operation will extend the file by one byte, and that byte will be locked. The unlock operation that follows has the effect of removing the locks for future writes that append data to the file, but it leaves a lock on the last byte in the file. When the second write occurs, the end of file is extended by one byte, but this byte is not locked. The state of the file locks for this sequence of steps is shown in Figure 14.10.

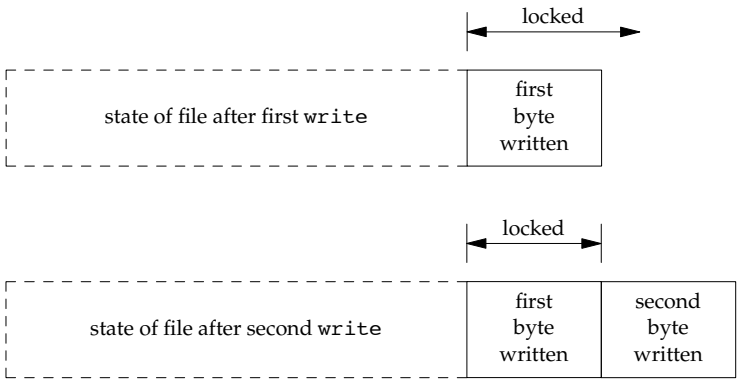


Figure 14.10 File range lock diagram

When a portion of a file is locked, the kernel converts the offset specified into an absolute file offset. In addition to specifying an absolute file offset (`SEEK_SET`), `fcntl` allows us to specify this offset relative to a point in the file: current (`SEEK_CUR`) or end of file (`SEEK_END`). The kernel needs to remember the locks independent of the current file offset or end of file, because the current offset and end of file can change, and changes to these attributes shouldn't affect the state of existing locks.

If we intended to remove the lock covering the byte we wrote in the first write, we could have specified the length as –1. Negative length values represent the bytes before the specified offset.

Advisory versus Mandatory Locking

Consider a library of database access routines. If all the functions in the library handle record locking in a consistent way, then we say that any set of processes using these functions to access a database are *cooperating processes*. It is feasible for these database access functions to use advisory locking if they are the only ones being used to access the database. But advisory locking doesn't prevent some other process that has write permission for the database file from writing whatever it wants to the database file. This rogue process would be an uncooperating process, since it's not using the accepted method (the library of database functions) to access the database.

Mandatory locking causes the kernel to check every `open`, `read`, and `write` to verify that the calling process isn't violating a lock on the file being accessed. Mandatory locking is sometimes called *enforcement-mode locking*.

We saw in Figure 14.2 that Linux 3.2.0 and Solaris 10 provide mandatory record locking, but FreeBSD 8.0 and Mac OS X 10.6.8 do not. Mandatory record locking is not part of the Single UNIX Specification. On Linux, if you want mandatory locking, you need to enable it on a per file system basis by using the `-o mand` option to the `mount` command.

Mandatory locking is enabled for a particular file by turning on the set-group-ID bit and turning off the group-execute bit. (Recall Figure 4.12.) Since the set-group-ID bit makes no sense when the group-execute bit is off, the designers of SVR3 chose this way to specify that the locking for a file is to be mandatory locking and not advisory locking.

What happens to a process that tries to `read` or `write` a file that has mandatory locking enabled and that part of the file is currently locked by another process? The answer depends on the type of operation (`read` or `write`), the type of lock held by the other process (read lock or write lock), and whether the descriptor for the `read` or `write` is nonblocking. Figure 14.11 shows the eight possibilities.

Type of existing lock on region held by other process	Blocking descriptor, tries to		Nonblocking descriptor, tries to	
	`read`	`write`	`read`	`write`
read lock	OK	blocks	OK	`EAGAIN`
write lock	blocks	blocks	`EAGAIN`	`EAGAIN`

Figure 14.11 Effect of mandatory locking on `read`s and `write`s by other processes

In addition to the `read` and `write` functions in Figure 14.11, the `open` function can be affected by mandatory record locks held by another process. Normally, `open` succeeds, even if the file being opened has outstanding mandatory record locks. The next `read` or `write` follows the rules listed in Figure 14.11. But if the file being opened has outstanding mandatory record locks (either read locks or write locks), and if the flags in the call to `open` specify either `O_TRUNC` or `O_CREAT`, then `open` returns an error of `EAGAIN` immediately, regardless of whether `O_NONBLOCK` is specified.

Only Solaris treats the `O_CREAT` flag as an error case. Linux allows the `O_CREAT` flag to be specified when opening a file with an outstanding mandatory lock. Generating the `open` error for `O_TRUNC` makes sense, because the file cannot be truncated if it is read locked or write locked by another process. Generating the error for `O_CREAT`, however, makes little sense; this flag says to create the file only if it doesn't already exist, but it has to exist to be record locked by another process.

This handling of locking conflicts with `open` can lead to surprising results. While developing the exercises in this section, a test program was run that opened a file (whose mode specified mandatory locking), established a read lock on an entire file, and then went to sleep for a while. (Recall from Figure 14.11 that a read lock should prevent writing to the file by other processes.) During this sleep period, the following behavior was seen in other typical UNIX System programs.

- The same file could be edited with the `ed` editor, and the results written back to disk! The mandatory record locking had no effect at all. Using the system call trace feature provided by some versions of the UNIX System, it was seen that `ed`

wrote the new contents to a temporary file, removed the original file, and then renamed the temporary file to be the original file. The mandatory record locking has no effect on the `unlink` function, which allowed this to happen.

> Under FreeBSD 8.0 and Solaris 10, we can obtain the system call trace of a process with the `truss`(1) command. Linux 3.2.0 provides the `strace`(1) command for the same purpose. Mac OS X 10.6.8 provides the `dtruss`(1m) command to trace system calls, but its use requires superuser privileges.

- The `vi` editor was never able to edit the file. It could read the file's contents, but whenever we tried to write new data to the file, `EAGAIN` was returned. If we tried to append new data to the file, the `write` blocked. This behavior from `vi` is what we expect.
- Using the Korn shell's `>` and `>>` operators to overwrite or append to the file resulted in the error "cannot create."
- Using the same two operators with the Bourne shell resulted in an error for `>`, but the `>>` operator just blocked until the mandatory lock was removed, and then proceeded. (The difference in the handling of the append operator occurs because the Korn shell `open`s the file with `O_CREAT` and `O_APPEND`, and we mentioned earlier that specifying `O_CREAT` generates an error. The Bourne shell, however, doesn't specify `O_CREAT` if the file already exists, so the `open` succeeds but the next `write` blocks.)

Results will vary, depending on the version of the operating system you are using. The bottom line, as demonstrated by this exercise, is to be wary of mandatory record locking. As seen with the `ed` example, it can be circumvented.

Mandatory record locking can also be used by a malicious user to hold a read lock on a file that is publicly readable. This can prevent anyone from writing to the file. (Of course, the file has to have mandatory record locking enabled for this to occur, which may require the user to be able to change the permission bits of the file.) Consider a database file that is world readable and has mandatory record locking enabled. If a malicious user were to hold a read lock on the entire file, the file could not be written to by other processes.

Example

We can run the program in Figure 14.12 to determine whether our system supports mandatory locking.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int             fd;
```

```
    pid_t           pid;
    char            buf[5];
    struct stat     statbuf;

    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");

    /* turn on set-group-ID and turn off group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        /* write lock entire file */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("write_lock error");

        TELL_CHILD(pid);

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
    } else {                /* child */
        WAIT_PARENT();      /* wait for parent to set lock */

        set_fl(fd, O_NONBLOCK);

        /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1)    /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n",
          errno);

        /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buf, 2) < 0)
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buf = %2.2s\n",
              buf);
    }
    exit(0);
}
```

Figure 14.12 Determine whether mandatory locking is supported

This program creates a file and enables mandatory locking for the file. The program then splits into parent and child, with the parent obtaining a write lock on the entire file. The child first sets its descriptor to be nonblocking and then attempts to obtain a read lock on the file, expecting to get an error. This lets us see whether the system returns `EACCES` or `EAGAIN`. Next, the child rewinds the file and tries to `read` from the file. If mandatory locking is provided, the `read` should return `EACCES` or `EAGAIN` (since the descriptor is nonblocking). Otherwise, the `read` returns the data that it read. Running this program under Solaris 10 (which supports mandatory locking) gives us

```
$ ./a.out temp.lock
read_lock of already-locked region returns 11
read failed (mandatory locking works): Resource temporarily unavailable
```

If we look at either the system's headers or the `intro`(2) manual page, we see that an `errno` of 11 corresponds to `EAGAIN`. Under FreeBSD 8.0, we get

```
$ ./a.out temp.lock
read_lock of already-locked region returns 35
read OK (no mandatory locking), buf = ab
```

Here, an `errno` of 35 corresponds to `EAGAIN`. Mandatory locking is not supported. □

Example

Let's return to the first question posed in this section: what happens when two people edit the same file at the same time? The normal UNIX System text editors do not use record locking, so the answer is still that the final result of the file corresponds to the last process that wrote the file.

Some versions of the `vi` editor use advisory record locking. Even if we were using one of these versions of `vi`, it still doesn't prevent users from running another editor that doesn't use advisory record locking.

If the system provides mandatory record locking, we could modify our favorite editor to use it (if we have the editor's source code). Not having the source code for the editor, we might try the following. We write our own program that is a front end to `vi`. This program immediately calls `fork`, and the parent just waits for the child to complete. The child opens the file specified on the command line, enables mandatory locking, obtains a write lock on the entire file, and then executes `vi`. While `vi` is running, the file is write locked, so other users can't modify it. When `vi` terminates, the parent's `wait` returns and our front end terminates.

A small front-end program of this type can be written, but it doesn't work. The problem is that it is common practice for editors to read their input file and then close it. A lock is released on a file whenever a descriptor that references that file is closed. As a result, when the editor closes the file after reading its contents, the lock is gone. There is no way to prevent this from happening in the front-end program. □

We'll use record locking in Chapter 20 in our database library to provide concurrent access to multiple processes. We'll also provide some timing measurements to see how record locking affects a process.

14.4 I/O Multiplexing

When we read from one descriptor and write to another, we can use blocking I/O in a loop, such as

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");
```

We see this form of blocking I/O over and over again. What if we have to read from two descriptors? In this case, we can't do a blocking `read` on either descriptor, as data may appear on one descriptor while we're blocked in a `read` on the other. A different technique is required to handle this case.

Let's look at the structure of the `telnet`(1) command. In this program, we read from the terminal (standard input) and write to a network connection, and we read from the network connection and write to the terminal (standard output). At the other end of the network connection, the `telnetd` daemon reads what we typed and presents it to a shell as if we were logged in to the remote machine. The `telnetd` daemon sends any output generated by the commands we type back to us through the `telnet` command, to be displayed on our terminal. Figure 14.13 shows a picture of this arrangement.

Figure 14.13 Overview of `telnet` program

The `telnet` process has two inputs and two outputs. We can't do a blocking `read` on either of the inputs, as we never know which input will have data for us.

One way to handle this particular problem is to divide the process in two pieces (using `fork`), with each half handling one direction of data. We show this in Figure 14.14. (The `cu`(1) command provided with System V's `uucp` communication package was structured like this.)

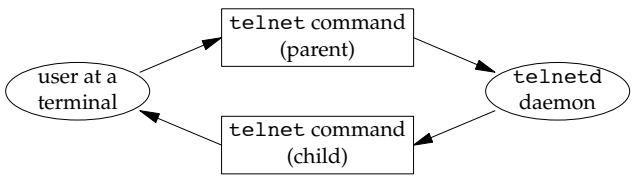


Figure 14.14 The `telnet` program using two processes

If we use two processes, we can let each process do a blocking `read`. But this leads to a problem when the operation terminates. If an end of file is received by the child (the

network connection is disconnected by the `telnetd` daemon), then the child terminates and the parent is notified by the `SIGCHLD` signal. But if the parent terminates (the user enters an end-of-file character at the terminal), then the parent has to tell the child to stop. We can use a signal for this (`SIGUSR1`, for example), but it does complicate the program somewhat.

Instead of two processes, we could use two threads in a single process. This avoids the termination complexity, but requires that we deal with synchronization between the threads, which could add more complexity than it saves.

We could use nonblocking I/O in a single process by setting both descriptors to be nonblocking and issuing a `read` on the first descriptor. If data is present, we read it and process it. If there is no data to read, the call returns immediately. We then do the same thing with the second descriptor. After this, we wait for some amount of time (a few seconds, perhaps) and then try to read from the first descriptor again. This type of loop is called *polling*. The problem is that it wastes CPU time. Most of the time, there won't be data to read, so we waste time performing the `read` system calls. We also have to guess how long to wait each time around the loop. Although it works on any system that supports nonblocking I/O, polling should be avoided on a multitasking system.

Another technique is called *asynchronous I/O*. With this technique, we tell the kernel to notify us with a signal when a descriptor is ready for I/O. There are two problems with this approach. First, although systems provide their own limited forms of asynchronous I/O, POSIX chose to standardize a different set of interfaces, so portability can be an issue. (In the past, POSIX asynchronous I/O was an optional facility in the Single UNIX Specification, but these interfaces are required as of SUSv4.) System V provides the `SIGPOLL` signal to support a limited form of asynchronous I/O, but this signal works only if the descriptor refers to a STREAMS device. BSD has a similar signal, `SIGIO`, but it has similar limitations: it works only on descriptors that refer to terminal devices or networks.

The second problem with this technique is that the limited forms use only one signal per process (`SIGPOLL` or `SIGIO`). If we enable this signal for two descriptors (in the example we've been talking about, reading from two descriptors), the occurrence of the signal doesn't tell us which descriptor is ready. Although the POSIX.1 asynchronous I/O interfaces allow us to select which signal to use for notification, the number of signals we can use is still far less than the number of possible open file descriptors. To determine which descriptor is ready, we would need to set each file descriptor to nonblocking mode and try the descriptors in sequence. We discuss asynchronous I/O in Section 14.5.

A better technique is to use *I/O multiplexing*. To do this, we build a list of the descriptors that we are interested in (usually more than one descriptor) and call a function that doesn't return until one of the descriptors is ready for I/O. Three functions—`poll`, `pselect`, and `select`—allow us to perform I/O multiplexing. On return from these functions, we are told which descriptors are ready for I/O.

> POSIX specifies that `<sys/select.h>` be included to pull the information for `select` into your program. Older systems require that you include `<sys/types.h>`, `<sys/time.h>`, and `<unistd.h>`. Check the `select` manual page to see what your system supports.

I/O multiplexing was provided with the `select` function in 4.2BSD. This function has always worked with any descriptor, although its main use has been for terminal I/O and network I/O. SVR3 added the `poll` function when the STREAMS mechanism was added. Initially, `poll` worked only with STREAMS devices. In SVR4, support was added to allow `poll` to work on any descriptor.

14.4.1 `select` and `pselect` Functions

The `select` function lets us do I/O multiplexing under all POSIX-compatible platforms. The arguments we pass to `select` tell the kernel

- Which descriptors we're interested in.
- Which conditions we're interested in for each descriptor. (Do we want to read from a given descriptor? Do we want to write to a given descriptor? Are we interested in an exception condition for a given descriptor?)
- How long we want to wait. (We can wait forever, wait a fixed amount of time, or not wait at all.)

On the return from `select`, the kernel tells us

- The total count of the number of descriptors that are ready
- Which descriptors are ready for each of the three conditions (read, write, or exception condition)

With this return information, we can call the appropriate I/O function (usually `read` or `write`) and know that the function won't block.

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

int select(int maxfdp1, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict exceptfds,
           struct timeval *restrict tvptr);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

Let's look at the last argument first. It specifies how long we want to wait in terms of seconds and microseconds (recall Section 4.20). There are three conditions.

tvptr `== NULL`

Wait forever. This infinite wait can be interrupted if we catch a signal. Return is made when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `select` returns –1 with `errno` set to `EINTR`.

tvptr->tv_sec `==` 0 `&&` *tvptr->tv_usec* `==` 0

Don't wait at all. All the specified descriptors are tested, and return is made immediately. This is a way to poll the system to find out the status of multiple descriptors without blocking in the `select` function.

tvptr->tv_sec `!=` 0 || *tvptr->tv_usec* `!=` 0

Wait the specified number of seconds and microseconds. Return is made when one of the specified descriptors is ready or when the timeout value expires. If the timeout expires before any of the descriptors is ready, the return value is 0. (If the system doesn't provide microsecond resolution, the *tvptr->tv_usec* value is rounded up to the nearest supported value.) As with the first condition, this wait can also be interrupted by a caught signal.

POSIX.1 allows an implementation to modify the `timeval` structure, so after `select` returns, you can't rely on the structure containing the same values it did before calling `select`. FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 all leave the structure unchanged, but Linux 3.2.0 will update it with the time remaining if `select` returns before the timeout value expires.

The middle three arguments—*readfds*, *writefds*, and *exceptfds*—are pointers to *descriptor sets*. These three sets specify which descriptors we're interested in and for which conditions (readable, writable, or an exception condition). A descriptor set is stored in an `fd_set` data type. This data type is chosen by the implementation so that it can hold one bit for each possible descriptor. We can consider it to be just a big array of bits, as shown in Figure 14.15.

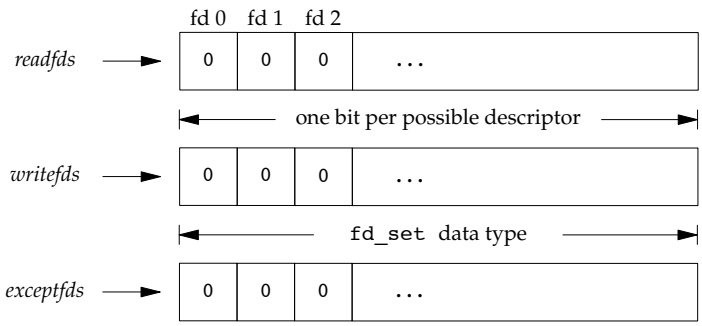


Figure 14.15 Specifying the read, write, and exception descriptors for `select`

The only thing we can do with the `fd_set` data type is allocate a variable of this type, assign a variable of this type to another variable of the same type, or use one of the following four functions on a variable of this type.

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

int FD_ISSET(int fd, fd_set *fdset);

                                        Returns: nonzero if fd is in set, 0 otherwise

void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

These interfaces can be implemented as either macros or functions. An `fd_set` is set to all zero bits by calling `FD_ZERO`. To turn on a single bit in a set, we use `FD_SET`. We can clear a single bit by calling `FD_CLR`. Finally, we can test whether a given bit is turned on in the set with `FD_ISSET`.

After declaring a descriptor set, we must zero the set using `FD_ZERO`. We then set bits in the set for each descriptor that we're interested in, as in

```
fd_set   rset;
int      fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

On return from `select`, we can test whether a given bit in the set is still on using `FD_ISSET`:

```
if (FD_ISSET(fd, &rset)) {
      .
      .
      .
}
```

Any (or all) of the middle three arguments to `select` (the pointers to the descriptor sets) can be null pointers if we're not interested in that condition. If all three pointers are `NULL`, then we have a higher-precision timer than is provided by `sleep`. (Recall from Section 10.19 that `sleep` waits for an integral number of seconds. With `select`, we can wait for intervals less than one second; the actual resolution depends on the system's clock.) Exercise 14.5 shows such a function.

The first argument to `select`, *maxfdp1*, stands for "maximum file descriptor plus 1." We calculate the highest descriptor that we're interested in, considering all three of the descriptor sets, add 1, and that's the first argument. We could just set the first argument to `FD_SETSIZE`, a constant in `<sys/select.h>` that specifies the maximum number of descriptors (often 1,024), but this value is too large for most applications. Indeed, most applications probably use between 3 and 10 descriptors. (Some applications need many more descriptors, but these UNIX programs are atypical.) By specifying the highest descriptor that we're interested in, we can prevent the kernel from going through hundreds of unused bits in the three descriptor sets, looking for bits that are turned on.

As an example, Figure 14.16 shows what two descriptor sets look like if we write

```
fd_set  readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

The reason we have to add 1 to the maximum descriptor number is that descriptors start at 0, and the first argument is really a count of the number of descriptors to check (starting with descriptor 0).

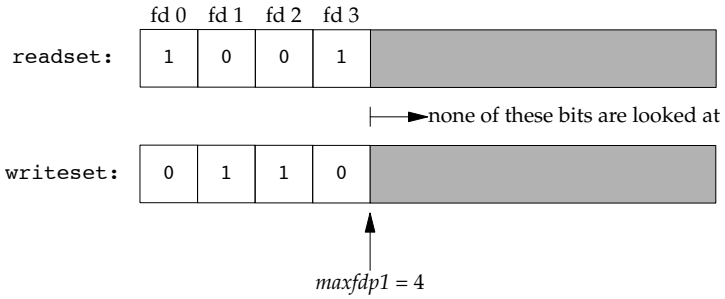


Figure 14.16 Example descriptor sets for `select`

There are three possible return values from `select`.

1. A return value of –1 means that an error occurred. This can happen, for example, if a signal is caught before any of the specified descriptors are ready. In this case, none of the descriptor sets will be modified.
2. A return value of 0 means that no descriptors are ready. This happens if the time limit expires before any of the descriptors are ready. When this happens, all the descriptor sets will be zeroed out.
3. A positive return value specifies the number of descriptors that are ready. This value is the sum of the descriptors ready in all three sets, so if the same descriptor is ready to be read *and* written, it will be counted twice in the return value. The only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready.

We now need to be more specific about what ''ready'' means.

- A descriptor in the read set (*readfds*) is considered ready if a `read` from that descriptor won't block.
- A descriptor in the write set (*writefds*) is considered ready if a `write` to that descriptor won't block.
- A descriptor in the exception set (*exceptfds*) is considered ready if an exception condition is pending on that descriptor. Currently, an exception condition corresponds to either the arrival of out-of-band data on a network connection or certain conditions occurring on a pseudo terminal that has been placed into packet mode. (Section 15.10 of Stevens [1990] describes this latter condition.)
- File descriptors for regular files always return ready for reading, writing, and exception conditions.

It is important to realize that whether a descriptor is blocking or not doesn't affect whether `select` blocks. That is, if we have a nonblocking descriptor that we want to read from and we call `select` with a timeout value of 5 seconds, `select` will block for up to 5 seconds. Similarly, if we specify an infinite timeout, `select` blocks until data is ready for the descriptor or until a signal is caught.

If we encounter the end of file on a descriptor, that descriptor is considered readable by `select`. We then call `read` and it returns 0—the way to signify end of file on UNIX systems. (Many people incorrectly assume that `select` indicates an exception condition on a descriptor when the end of file is reached.)

POSIX.1 also defines a variant of `select` called `pselect`.

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

int pselect(int maxfdp1, fd_set *restrict readfds,
            fd_set *restrict writefds, fd_set *restrict exceptfds,
            const struct timespec *restrict tsptr,
            const sigset_t *restrict sigmask);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

The `pselect` function is identical to `select`, with the following exceptions.

- The timeout value for `select` is specified by a `timeval` structure, but for `pselect`, a `timespec` structure is used. (Recall the definition of the `timespec` structure in Section 4.2.) Instead of seconds and microseconds, the `timespec` structure represents the timeout value in seconds and nanoseconds. This provides a higher-resolution timeout if the platform supports that fine a level of granularity.
- The timeout value for `pselect` is declared `const`, and we are guaranteed that its value will not change as a result of calling `pselect`.
- An optional signal mask argument is available with `pselect`. If *sigmask* is `NULL`, `pselect` behaves as `select` does with respect to signals. Otherwise, *sigmask* points to a signal mask that is atomically installed when `pselect` is called. On return, the previous signal mask is restored.

14.4.2 `poll` Function

The `poll` function is similar to `select`, but the programmer interface is different. This function was originally introduced in System V to support the STREAMS subsystem, but we are able to use it with any type of file descriptor.

```
#include <poll.h>

int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

With poll, instead of building a set of descriptors for each condition (readability, writability, and exception condition) as we did with select, we build an array of pollfd structures, with each array element specifying a descriptor number and the conditions that we're interested in for that descriptor:

```
struct pollfd {
  int    fd;       /* file descriptor to check, or <0 to ignore */
  short  events;   /* events of interest on fd */
  short  revents;  /* events that occurred on fd */
};
```

The number of elements in the *fdarray* array is specified by *nfds*.

> Historically, there have been differences in how the *nfds* parameter was declared. SVR3 specified the number of elements in the array as an unsigned long, which seems excessive. In the SVR4 manual [AT&T 1990d], the prototype for poll showed the data type of the second argument as size_t. (Recall the primitive system data types from Figure 2.21.) But the actual prototype in the <poll.h> header still showed the second argument as an unsigned long. The Single UNIX Specification defines the new type nfds_t to allow the implementation to select the appropriate type and hide the details from applications. Note that this type has to be large enough to hold an integer, since the return value represents the number of entries in the array with satisfied events.
>
> The SVID corresponding to SVR4 [AT&T 1989] showed the first argument to poll as struct pollfd *fdarray*[], whereas the SVR4 manual page [AT&T 1990d] showed this argument as struct pollfd **fdarray*. In the C language, both declarations are equivalent. We use the first declaration to reiterate that fdarray points to an array of structures and not a pointer to a single structure.

To tell the kernel which events we're interested in for each descriptor, we have to set the events member of each array element to one or more of the values in Figure 14.17. On return, the revents member is set by the kernel, thereby specifying which events have occurred for each descriptor. (Note that poll doesn't change the events member. This behavior differs from that of select, which modifies its arguments to indicate what is ready.)

Name	Input to events?	Result from revents?	Description
POLLIN	•	•	Data other than high priority data can be read without blocking (equivalent to POLLRDNORM | POLLRDBAND).
POLLRDNORM	•	•	Normal data can be read without blocking.
POLLRDBAND	•	•	Priority data can be read without blocking.
POLLPRI	•	•	High-priority data can be read without blocking.
POLLOUT	•	•	Normal data can be written without blocking.
POLLWRNORM	•	•	Same as POLLOUT.
POLLWRBAND	•	•	Priority data can be written without blocking.
POLLERR		•	An error has occurred.
POLLHUP		•	A hangup has occurred.
POLLNVAL		•	The descriptor does not reference an open file.

Figure 14.17 The events and revents flags for poll

The first four rows of Figure 14.17 test for readability, the next three test for writability, and the final three are for exception conditions. The last three rows in Figure 14.17 are set by the kernel on return. These three values are returned in `revents` when the condition occurs, even if they weren't specified in the `events` field.

> The poll event names containing the term *BAND* refer to priority bands in STREAMS. Refer to Rago [1993] for more information about STREAMS and priority bands.

When a descriptor is hung up (`POLLHUP`), we can no longer write to the descriptor. There may, however, still be data to be read from the descriptor.

The final argument to `poll` specifies how long we want to wait. As with `select`, there are three cases.

timeout `==` –1
: Wait forever. (Some systems define the constant `INFTIM` in `<stropts.h>` as –1.) We return when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `poll` returns –1 with `errno` set to `EINTR`.

timeout `==` 0
: Don't wait. All the specified descriptors are tested, and we return immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the call to `poll`.

timeout `>` 0
: Wait *timeout* milliseconds. We return when one of the specified descriptors is ready or when the *timeout* expires. If the *timeout* expires before any of the descriptors is ready, the return value is 0. (If your system doesn't provide millisecond resolution, *timeout* is rounded up to the nearest supported value.)

It is important to realize the difference between an end of file and a hangup. If we're entering data from the terminal and type the end-of-file character, `POLLIN` is turned on so we can read the end-of-file indication (`read` returns 0). `POLLHUP` is not turned on in `revents`. If we're reading from a modem and the telephone line is hung up, we'll receive the `POLLHUP` notification.

As with `select`, whether a descriptor is blocking doesn't affect whether `poll` blocks.

Interruptibility of `select` and `poll`

When the automatic restarting of interrupted system calls was introduced with 4.2BSD (Section 10.5), the `select` function was never restarted. This characteristic continues with most systems even if the `SA_RESTART` option is specified. But under SVR4, if `SA_RESTART` was specified, even `select` and `poll` were automatically restarted. To prevent this from catching us when we port software to systems derived from SVR4, we'll always use the `signal_intr` function (Figure 10.19) if the signal could interrupt a call to `select` or `poll`.

None of the implementations described in this book restart `poll` or `select` when a signal is received, even if the `SA_RESTART` flag is used.

14.5 Asynchronous I/O

Using `select` and `poll`, as described in the previous section, is a synchronous form of notification. The system doesn't tell us anything until we ask (by calling either `select` or `poll`). As we saw in Chapter 10, signals provide an asynchronous form of notification that something has happened. All systems derived from BSD and System V provide some form of asynchronous I/O, using a signal (`SIGPOLL` in System V; `SIGIO` in BSD) to notify the process that something of interest has happened on a descriptor. As mentioned in the previous section, these forms of asynchronous I/O are limited: they don't work with all file types and they allow the use of only one signal. If we enable more than one descriptor for asynchronous I/O, we cannot tell which descriptor the signal corresponds to when the signal is delivered.

Version 4 of the Single UNIX Specification moved the general asynchronous I/O mechanism from the real-time extensions to the base specification. This mechanism addresses the limitations that exist with these older asynchronous I/O facilities.

Before we look at the different ways to use asynchronous I/O, we need to discuss the costs. When we decide to use asynchronous I/O, we complicate the design of our application by choosing to juggle multiple concurrent operations. A simpler approach may be to use multiple threads, which would allow us to write the program using a synchronous model, and let the threads run asynchronous to each other.

We incur additional complexity when we use the POSIX asynchronous I/O interfaces:

- We have to worry about three sources of errors for every asynchronous operation: one associated with the submission of the operation, one associated with the result of the operation itself, and one associated with the functions used to determine the status of the asynchronous operations.
- The interfaces themselves involve a lot of extra setup and processing rules compared to their conventional counterparts, as we shall see.

  We can't really call the non-asynchronous I/O function calls "synchronous," because although they are synchronous with respect to the program flow, they aren't synchronous with respect to the I/O. Recall the discussion of synchronous writes in Chapter 3. We call a write "synchronous" if the data we write is persistent when we return from the call to the `write` function. We also can't differentiate the conventional I/O function calls from the asynchronous ones by referring to the conventional calls as the "standard" I/O calls, because this confuses them with the function calls in the standard I/O library. To avoid confusion, we'll refer to the `read` and `write` functions as the "conventional" I/O function calls in this section.

- Recovering from errors can be difficult. For example, if we submit multiple asynchronous writes and one fails, how should we proceed? If the writes are related, we might have to undo the ones that succeeded.

14.5.1 System V Asynchronous I/O

System V provides a limited form of asynchronous I/O that works only with STREAMS devices and STREAMS pipes. The System V asynchronous I/O signal is `SIGPOLL`.

To enable asynchronous I/O for a STREAMS device, we have to call `ioctl` with a second argument (*request*) of `I_SETSIG`. The third argument is an integer value formed from one or more of the constants in Figure 14.18. These constants are defined in `<stropts.h>`.

> Interfaces related to the STREAMS mechanism were marked obsolescent in SUSv4, so we don't cover them in any detail. See Rago [1993] for more information about STREAMS.

Constant	Description
`S_INPUT`	We can read data (other than high-priority data) without blocking.
`S_RDNORM`	We can read normal data without blocking.
`S_RDBAND`	We can read priority data without blocking.
`S_BANDURG`	If this constant is specified with `S_RDBAND`, the `SIGURG` signal is generated instead of `SIGPOLL` when we can read priority data without blocking.
`S_HIPRI`	We can read high-priority data without blocking.
`S_OUTPUT`	We can write normal data without blocking.
`S_WRNORM`	Same as `S_OUTPUT`.
`S_WRBAND`	We can write priority data without blocking.
`S_MSG`	The `SIGPOLL` signal message has reached the stream head.
`S_ERROR`	The stream has an error.
`S_HANGUP`	The stream has hung up.

Figure 14.18 Conditions for generating `SIGPOLL` signal

In addition to calling `ioctl` to specify the conditions that should generate the `SIGPOLL` signal, we have to establish a signal handler for this signal. Recall from Figure 10.1 that the default action for `SIGPOLL` is to terminate the process, so we should establish the signal handler before calling `ioctl`.

14.5.2 BSD Asynchronous I/O

Asynchronous I/O in BSD-derived systems is a combination of two signals: `SIGIO` and `SIGURG`. The former is the general asynchronous I/O signal, and the latter is used only to notify the process that out-of-band data has arrived on a network connection.

To receive the `SIGIO` signal, we need to perform three steps.

1. Establish a signal handler for `SIGIO`, by calling either `signal` or `sigaction`.
2. Set the process ID or process group ID to receive the signal for the descriptor, by calling `fcntl` with a command of `F_SETOWN` (Section 3.14).

3. Enable asynchronous I/O on the descriptor by calling `fcntl` with a command of `F_SETFL` to set the `O_ASYNC` file status flag (Figure 3.10).

Step 3 can be performed only on descriptors that refer to terminals or networks, which is a fundamental limitation of the BSD asynchronous I/O facility.

For the `SIGURG` signal, we need perform only steps 1 and 2. `SIGURG` is generated only for descriptors that refer to network connections that support out-of-band data, such as TCP connections.

14.5.3 POSIX Asynchronous I/O

The POSIX asynchronous I/O interfaces give us a consistent way to perform asynchronous I/O, regardless of the type of file. These interfaces were adopted from the real-time draft standard, which themselves were an option in the Single UNIX Specification. In Version 4, the Single UNIX Specification moved these interfaces to the base, so they are now required to be supported by all platforms.

The asynchronous I/O interfaces use AIO control blocks to describe I/O operations. The `aiocb` structure defines an AIO control block. It contains at least the fields shown in the following structure (implementations might include additional fields):

```
struct aiocb {
  int             aio_fildes;      /* file descriptor */
  off_t           aio_offset;      /* file offset for I/O */
  volatile void  *aio_buf;         /* buffer for I/O */
  size_t          aio_nbytes;      /* number of bytes to transfer */
  int             aio_reqprio;     /* priority */
  struct sigevent aio_sigevent;    /* signal information */
  int             aio_lio_opcode;  /* operation for list I/O */
};
```

The `aio_fildes` field is the file descriptor open for the file to be read or written. The read or write starts at the offset specified by `aio_offset`. For a read, data is copied to the buffer that begins at the address specified by `aio_buf`. For a write, data is copied from this buffer. The `aio_nbytes` field contains the number of bytes to read or write.

Note that we have to provide an explicit offset when we perform asynchronous I/O. The asynchronous I/O interfaces don't affect the file offset maintained by the operating system. This won't be a problem as long as we never mix asynchronous I/O functions with conventional I/O functions on the same file in a process. Also note that if we write to a file opened in append mode (with `O_APPEND`) using an asynchronous interface, the `aio_offset` field in the AIO control block is ignored by the system.

The other fields don't correspond to the conventional I/O functions. The `aio_reqprio` field is a hint that gives applications a way to suggest an ordering for the asynchronous I/O requests. The system has only limited control over the exact ordering, however, so there is no guarantee that the hint will be honored. The `aio_lio_opcode` field is used only with list-based asynchronous I/O, which we'll

discuss shortly. The aio_sigevent field controls how the application is notified about the completion of the I/O event. It is described by a sigevent structure.

```
struct sigevent {
  int             sigev_notify;                 /* notify type */
  int             sigev_signo;                  /* signal number */
  union sigval    sigev_value;                  /* notify argument */
  void (*sigev_notify_function)(union sigval); /* notify function */
  pthread_attr_t *sigev_notify_attributes;      /* notify attrs */
};
```

The sigev_notify field controls the type of notification. It can take on one of three values.

SIGEV_NONE The process is not notified when the asynchronous I/O request completes.

SIGEV_SIGNAL The signal specified by the sigev_signo field is generated when the asynchronous I/O request completes. If the application has elected to catch the signal and has specified the SA_SIGINFO flag when establishing the signal handler, the signal is queued (if the implementation supports queued signals). The signal handler is passed a siginfo structure whose si_value field is set to sigev_value (again, if SA_SIGINFO is used).

SIGEV_THREAD The function specified by the sigev_notify_function field is called when the asynchronous I/O request completes. It is passed the sigev_value field as its only argument. The function is executed in a separate thread in a detached state, unless the sigev_notify_attributes field is set to the address of a pthread attribute structure specifying alternative attributes for the thread.

To perform asynchronous I/O, we need to initialize an AIO control block and call either the aio_read function to make an asynchronous read or the aio_write function to make an asynchronous write.

```
#include <aio.h>

int aio_read(struct aiocb *aiocb);

int aio_write(struct aiocb *aiocb);
```

Both return: 0 if OK, –1 on error

When these functions return success, the asynchronous I/O request has been queued for processing by the operating system. The return value bears no relation to the result of the actual I/O operation. While the I/O operation is pending, we have to be careful to ensure that the AIO control block and data buffer remain stable; their underlying memory must remain valid and we can't reuse them until the I/O operation completes.

To force all pending asynchronous writes to persistent storage without waiting, we can set up an AIO control block and call the aio_fsync function.

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocb);
```

Returns: 0 if OK, –1 on error

The `aio_fildes` field in the AIO control block indicates the file whose asynchronous writes are synched. If the *op* argument is set to `O_DSYNC`, then the operation behaves like a call to `fdatasync`. Otherwise, if *op* is set to `O_SYNC`, the operation behaves like a call to `fsync`.

Like the `aio_read` and `aio_write` functions, the `aio_fsync` operation returns when the synch is scheduled. The data won't be persistent until the asynchronous synch completes. The AIO control block controls how we are notified, just as with the `aio_read` and `aio_write` functions.

To determine the completion status of an asynchronous read, write, or synch operation, we need to call the `aio_error` function.

```
#include <aio.h>

int aio_error(const struct aiocb *aiocb);
```

Returns: (see following)

The return value tells us one of four things.

`0`	The asynchronous operation completed successfully. We need to call the `aio_return` function to obtain the return value from the operation.
–1	The call to `aio_error` failed. In this case, `errno` tells us why.
`EINPROGRESS`	The asynchronous read, write, or synch is still pending.
anything else	Any other return value gives us the error code corresponding to the failed asynchronous operation.

If the asynchronous operation succeeded, we can call the `aio_return` function to get the asynchronous operation's return value.

```
#include <aio.h>

ssize_t aio_return(const struct aiocb *aiocb);
```

Returns: (see following)

Until the asynchronous operation completes, we need to be careful to avoid calling the `aio_return` function. The results are undefined until the operation completes. We also need to be careful to call `aio_return` only one time per asynchronous I/O operation. Once we call this function, the operating system is free to deallocate the record containing the I/O operation's return value.

The `aio_return` function will return –1 and set `errno` if `aio_return` itself fails. Otherwise, it will return the results of the asynchronous operation. In this case, it will return whatever `read`, `write`, or `fsync` would have returned on success if one of those functions had been called.

We use asynchronous I/O when we have other processing to do and we don't want to block while performing the I/O operation. However, when we have completed the processing and find that we still have asynchronous operations outstanding, we can call the `aio_suspend` function to block until an operation completes.

```
#include <aio.h>

int aio_suspend(const struct aiocb *const list[], int nent,
                const struct timespec *timeout);
```

Returns: 0 if OK, –1 on error

One of three things can cause `aio_suspend` to return. If we are interrupted by a signal, it returns –1 with `errno` set to `EINTR`. If the time limit specified by the optional *timeout* argument expires without any of the I/O operations completing, then `aio_suspend` returns –1 with `errno` set to `EAGAIN` (we can pass a null pointer for the *timeout* argument if we want to block without a time limit). If any of the I/O operations complete, `aio_suspend` returns 0. If all asynchronous I/O operations are complete when we call `aio_suspend`, then `aio_suspend` will return without blocking.

The *list* argument is a pointer to an array of AIO control blocks and the *nent* argument indicates the number of entries in the array. Null pointers in the array are skipped; the other entries must point to AIO control blocks that have been used to initiate asynchronous I/O operations.

When we have pending asynchronous I/O operations that we no longer want to complete, we can attempt to cancel them with the `aio_cancel` function.

```
#include <aio.h>

int aio_cancel(int fd, struct aiocb *aiocb);
```

Returns: (see following)

The *fd* argument specifies the file descriptor with the outstanding asynchronous I/O operations. If the *aiocb* argument is `NULL`, then the system attempts to cancel all outstanding asynchronous I/O operations on the file. Otherwise, the system attempts to cancel the single asynchronous I/O operation described by the AIO control block. We say that the system "attempts" to cancel the operations, because there is no guarantee that the system will be able to cancel any operations that are in progress.

The `aio_cancel` function can return one of four values:

`AIO_ALLDONE`	All of the operations completed before the attempt to cancel them.
`AIO_CANCELED`	All of the requested operations have been canceled.
`AIO_NOTCANCELED`	At least one of the requested operations could not be canceled.
–1	The call to `aio_cancel` failed. The error code will be stored in `errno`.

If an asynchronous I/O operation is successfully canceled, calling the `aio_error` function on the corresponding AIO control block will return the error `ECANCELED`. If the operation can't be canceled, then the corresponding AIO control block is unchanged by the call to `aio_cancel`.

One additional function is included with the asynchronous I/O interfaces, although it can be used in either a synchronous or an asynchronous manner. The `lio_listio` function submits a set of I/O requests described by a list of AIO control blocks.

```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
               int nent, struct sigevent *restrict sigev);
```

Returns: 0 if OK, –1 on error

The *mode* argument determines whether the I/O is truly asynchronous. When it is set to `LIO_WAIT`, the `lio_listio` function won't return until all of the I/O operations specified by the list are complete. In this case, the *sigev* argument is ignored. When the *mode* argument is set to `LIO_NOWAIT`, then the `lio_listio` function returns as soon as the I/O requests are queued. The process is notified asynchronously when all of the I/O operations complete, as specified by the *sigev* argument. If we don't want to be notified, we can set *sigev* to `NULL`. Note that the individual AIO control blocks themselves may also enable asynchronous notification when an individual operation completes. The asynchronous notification specified by the *sigev* argument is in addition to these, and is sent only when all of the I/O operations complete.

The *list* argument points to a list of AIO control blocks specifying the I/O operations to perform. The *nent* argument specifies the number of elements in the array. The list of AIO control blocks can contain `NULL` pointers; these entries are ignored.

In each AIO control block, the `aio_lio_opcode` field specifies whether the operation is a read (`LIO_READ`), a write (`LIO_WRITE`), or a no-op (`LIO_NOP`), which is ignored. A read is treated as if the corresponding AIO control block had been passed to the `aio_read` function. Similarly, a write is treated as if the AIO control block had been passed to `aio_write`.

Implementations can limit the number of asynchronous I/O operations we are allowed to have outstanding. The limits are runtime invariants, and are summarized in Figure 14.19.

Name	Description	Minimum acceptable value
`AIO_LISTIO_MAX`	maximum number of I/O operations in a single list I/O call	`_POSIX_AIO_LISTIO_MAX (2)`
`AIO_MAX`	maximum number of outstanding asynchronous I/O operations	`_POSIX_AIO_MAX (1)`
`AIO_PRIO_DELTA_MAX`	maximum amount by which a process can decrease its asynchronous I/O priority level	0

Figure 14.19 POSIX.1 runtime invariant values for asynchronous I/O

We can determine the value of AIO_LISTIO_MAX by calling the sysconf function with the *name* argument set to _SC_IO_LISTIO_MAX. Similarly, we can determine the value of AIO_MAX by calling sysconf with the *name* argument set to _SC_AIO_MAX, and we can get the value of AIO_PRIO_DELTA_MAX by calling sysconf with its argument set to _SC_AIO_PRIO_DELTA_MAX.

The POSIX asynchronous I/O interfaces were originally introduced to provide real-time applications with a way to avoid being blocked while performing I/O operations. Now we'll look at an example of how to use the interfaces.

Example

We don't discuss real-time programming in this text, but because the POSIX asynchronous I/O interfaces are now part of the base specification in the Single UNIX Specification, we'll look at how to use them. To compare the asynchronous I/O interfaces with their conventional counterparts, we'll look at the task of translating a file from one format to another.

The program shown in Figure 14.20 translates a file using the ROT-13 algorithm that the USENET news system, popular in the 1980s, used to obscure text that might be offensive or contain spoilers or joke punchlines. The algorithm rotates the characters 'a' to 'z' and 'A' to 'Z' by 13 positions, but leaves all other characters unchanged.

```
#include "apue.h"
#include <ctype.h>
#include <fcntl.h>

#define BSZ 4096

unsigned char buf[BSZ];

unsigned char
translate(unsigned char c)
{
    if (isalpha(c)) {
        if (c >= 'n')
            c -= 13;
        else if (c >= 'a')
            c += 13;
        else if (c >= 'N')
            c -= 13;
        else
            c += 13;
    }
    return(c);
}

int
main(int argc, char* argv[])
{
    int ifd, ofd, i, n, nw;
```

```
    if (argc != 3)
        err_quit("usage: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
        err_sys("can't create %s", argv[2]);

    while ((n = read(ifd, buf, BSZ)) > 0) {
        for (i = 0; i < n; i++)
            buf[i] = translate(buf[i]);
        if ((nw = write(ofd, buf, n)) != n) {
            if (nw < 0)
                err_sys("write failed");
            else
                err_quit("short write (%d/%d)", nw, n);
        }
    }

    fsync(ofd);
    exit(0);
}
```

Figure 14.20 Translate a file using ROT-13

The I/O portion of the program is straightforward: we read a block from the input file, translate it, and then write the block to the output file. We repeat this until we hit the end of file and `read` returns zero. The program in Figure 14.21 shows how to perform the same task using the equivalent asynchronous I/O functions.

```
#include "apue.h"
#include <ctype.h>
#include <fcntl.h>
#include <aio.h>
#include <errno.h>

#define BSZ 4096
#define NBUF 8

enum rwop {
    UNUSED = 0,
    READ_PENDING = 1,
    WRITE_PENDING = 2
};

struct buf {
    enum rwop     op;
    int           last;
    struct aiocb  aiocb;
    unsigned char data[BSZ];
};

struct buf bufs[NBUF];
```

```
unsigned char
translate(unsigned char c)
{
    /* same as before */
}

int
main(int argc, char* argv[])
{
    int                 ifd, ofd, i, j, n, err, numop;
    struct stat         sbuf;
    const struct aiocb  *aiolist[NBUF];
    off_t               off = 0;

    if (argc != 3)
        err_quit("usage: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
        err_sys("can't create %s", argv[2]);
    if (fstat(ifd, &sbuf) < 0)
        err_sys("fstat failed");

    /* initialize the buffers */
    for (i = 0; i < NBUF; i++) {
        bufs[i].op = UNUSED;
        bufs[i].aiocb.aio_buf = bufs[i].data;
        bufs[i].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
        aiolist[i] = NULL;
    }

    numop = 0;
    for (;;) {
        for (i = 0; i < NBUF; i++) {
            switch (bufs[i].op) {
            case UNUSED:
                /*
                 * Read from the input file if more data
                 * remains unread.
                 */
                if (off < sbuf.st_size) {
                    bufs[i].op = READ_PENDING;
                    bufs[i].aiocb.aio_fildes = ifd;
                    bufs[i].aiocb.aio_offset = off;
                    off += BSZ;
                    if (off >= sbuf.st_size)
                        bufs[i].last = 1;
                    bufs[i].aiocb.aio_nbytes = BSZ;
                    if (aio_read(&bufs[i].aiocb) < 0)
                        err_sys("aio_read failed");
                    aiolist[i] = &bufs[i].aiocb;
```

```
            numop++;
        }
        break;

    case READ_PENDING:
        if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
            continue;
        if (err != 0) {
            if (err == -1)
                err_sys("aio_error failed");
            else
                err_exit(err, "read failed");
        }

        /*
         * A read is complete; translate the buffer
         * and write it.
         */
        if ((n = aio_return(&bufs[i].aiocb)) < 0)
            err_sys("aio_return failed");
        if (n != BSZ && !bufs[i].last)
            err_quit("short read (%d/%d)", n, BSZ);
        for (j = 0; j < n; j++)
            bufs[i].data[j] = translate(bufs[i].data[j]);
        bufs[i].op = WRITE_PENDING;
        bufs[i].aiocb.aio_fildes = ofd;
        bufs[i].aiocb.aio_nbytes = n;
        if (aio_write(&bufs[i].aiocb) < 0)
            err_sys("aio_write failed");
        /* retain our spot in aiolist */
        break;

    case WRITE_PENDING:
        if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
            continue;
        if (err != 0) {
            if (err == -1)
                err_sys("aio_error failed");
            else
                err_exit(err, "write failed");
        }

        /*
         * A write is complete; mark the buffer as unused.
         */
        if ((n = aio_return(&bufs[i].aiocb)) < 0)
            err_sys("aio_return failed");
        if (n != bufs[i].aiocb.aio_nbytes)
            err_quit("short write (%d/%d)", n, BSZ);
        aiolist[i] = NULL;
        bufs[i].op = UNUSED;
```

```
                numop--;
                break;
            }
        }
        if (numop == 0) {
            if (off >= sbuf.st_size)
                break;
        } else {
            if (aio_suspend(aiolist, NBUF, NULL) < 0)
                err_sys("aio_suspend failed");
        }
    }

    bufs[0].aiocb.aio_fildes = ofd;
    if (aio_fsync(O_SYNC, &bufs[0].aiocb) < 0)
        err_sys("aio_fsync failed");
    exit(0);
}
```

Figure 14.21 Translate a file using ROT-13 and asynchronous I/O

Note that we use eight buffers, so we can have up to eight asynchronous I/O requests pending. Surprisingly, this might actually reduce performance—if the reads are presented to the file system out of order, it can defeat the operating system's read-ahead algorithm.

Before we can check the return value of an operation, we need to make sure the operation has completed. When `aio_error` returns a value other than `EINPROGRESS` or –1, we know the operation is complete. Excluding these values, if the return value is anything other than 0, then we know the operation failed. Once we've checked these conditions, it is safe to call `aio_return` to get the return value of the I/O operation.

As long as we have work to do, we can submit asynchronous I/O operations. When we have an unused AIO control block, we can submit an asynchronous read request. When a read completes, we translate the buffer contents and then submit an asynchronous write request. When all AIO control blocks are in use, we wait for an operation to complete by calling `aio_suspend`.

When we write a block to the output file, we retain the same offset at which we read the data from the input file. Consequently, the order of the writes doesn't matter. This strategy works only because each character in the input file has a corresponding character in the output file at the same offset; we neither add nor delete characters in the output file. (This insight might help solve Exercise 14.8.)

We don't use asynchronous notification in this example, because it is easier to use a synchronous programming model. If we had something else to do while the I/O operations were in progress, then the additional work could be folded into the `for` loop. If we needed to prevent this additional work from delaying the task of translating the file, however, then we might have to structure the code to use some form of asynchronous notification. With multiple tasks, we need to prioritize the tasks before deciding how the program should be structured. □

14.6 readv and writev Functions

The readv and writev functions let us read into and write from multiple noncontiguous buffers in a single function call. These operations are called *scatter read* and *gather write*.

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

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);

ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Both return: number of bytes read or written, –1 on error

The second argument to both functions is a pointer to an array of iovec structures:

```
struct iovec {
  void   *iov_base;  /* starting address of buffer */
  size_t  iov_len;   /* size of buffer */
};
```

The number of elements in the *iov* array is specified by *iovcnt*. It is limited to IOV_MAX (recall Figure 2.11). Figure 14.22 shows a diagram relating the arguments to these two functions and the iovec structure.

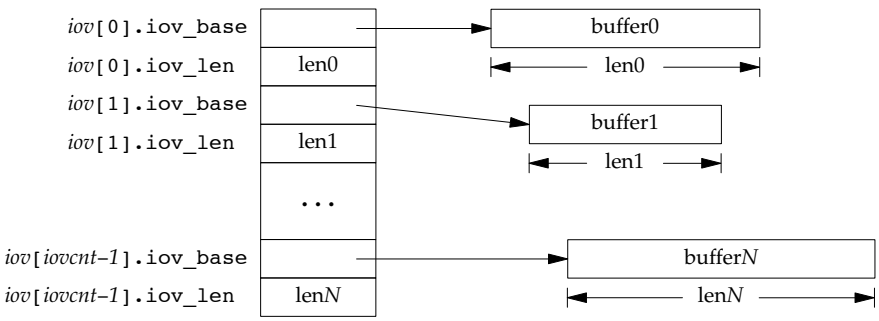


Figure 14.22 The iovec structure for readv and writev

The writev function gathers the output data from the buffers in order: *iov[0]*, *iov[1]*, through *iov[iovcnt–1]*; writev returns the total number of bytes output, which should normally equal the sum of all the buffer lengths.

The readv function scatters the data into the buffers in order, always filling one buffer before proceeding to the next. readv returns the total number of bytes that were read. A count of 0 is returned if there is no more data and the end of file is encountered.

These two functions originated in 4.2BSD and were later added to SVR4. These two functions are included in the XSI option of the Single UNIX Specification.