Format	Description	Example
`%a`	abbreviated weekday name	`Tue`
`%A`	full weekday name	`Tuesday`
`%b`	abbreviated month name	`Feb`
`%B`	full month name	`February`
`%c`	date and time	`Tue Feb 10 18:27:38 2004`
`%C`	year/100: [00–99]	`20`
`%d`	day of the month: [01–31]	`10`
`%D`	date [MM/DD/YY]	`02/10/04`
`%e`	day of month (single digit preceded by space) [1–31]	`10`
`%F`	ISO 8601 date format [YYYY-MM-DD]	`2004-02-10`
`%g`	last two digits of ISO 8601 week-based year [00–99]	`04`
`%G`	ISO 8601 week-based year	`2004`
`%h`	same as %b	`Feb`
`%H`	hour of the day (24-hour format): [00–23]	`18`
`%I`	hour of the day (12-hour format): [01–12]	`06`
`%j`	day of the year: [001–366]	`041`
`%m`	month: [01–12]	`02`
`%M`	minute: [00–59]	`27`
`%n`	newline character	
`%p`	AM/PM	`PM`
`%r`	locale's time (12-hour format)	`06:27:38 PM`
`%R`	same as "%H:%M"	`18:27`
`%S`	second: [00–60]	`38`
`%t`	horizontal tab character	
`%T`	same as "%H:%M:%S"	`18:27:38`
`%u`	ISO 8601 weekday [Monday=1, 1–7]	`2`
`%U`	Sunday week number: [00–53]	`06`
`%V`	ISO 8601 week number: [01–53]	`07`
`%w`	weekday: [0=Sunday, 0–6]	`2`
`%W`	Monday week number: [00–53]	`06`
`%x`	date	`02/10/04`
`%X`	time	`18:27:38`
`%y`	last two digits of year: [00–99]	`04`
`%Y`	year	`2004`
`%z`	offset from UTC in ISO 8601 format	`-0500`
`%Z`	time zone name	`EST`
`%%`	translates to a percent sign	`%`

Figure 6.9 Conversion specifiers for `strftime`

6.11 Summary

The password file and the group file are used on all UNIX systems. We've looked at the various functions that read these files. We've also talked about shadow passwords, which can help system security. Supplementary group IDs provide a way to participate in multiple groups at the same time. We also looked at how similar functions are provided by most systems to access other system-related data files. We discussed the

POSIX.1 functions that programs can use to identify the system on which they are running. We finished the chapter with a look at the time and date functions provided by ISO C and the Single UNIX Specification.

Exercises

6.1 If the system uses a shadow file and we need to obtain the encrypted password, how do we do it?

6.2 If you have superuser access and your system uses shadow passwords, implement the previous exercise.

6.3 Write a program that calls `uname` and prints all the fields in the `utsname` structure. Compare the output to the output from the `uname(1)` command.

6.4 Calculate the latest time that can be represented by the `time_t` data type. After it wraps around, what happens?

6.5 Write a program to obtain the current time and print it using `strftime`, so that it looks like the default output from `date(1)`. Set the `TZ` environment variable to different values and see what happens.

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 assembler) 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 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:

```
$ cc 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:

```
$ cc -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 up to 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. The `sysconf` function can be used to determine the maximum number of exit handlers supported by a given platform (see 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.

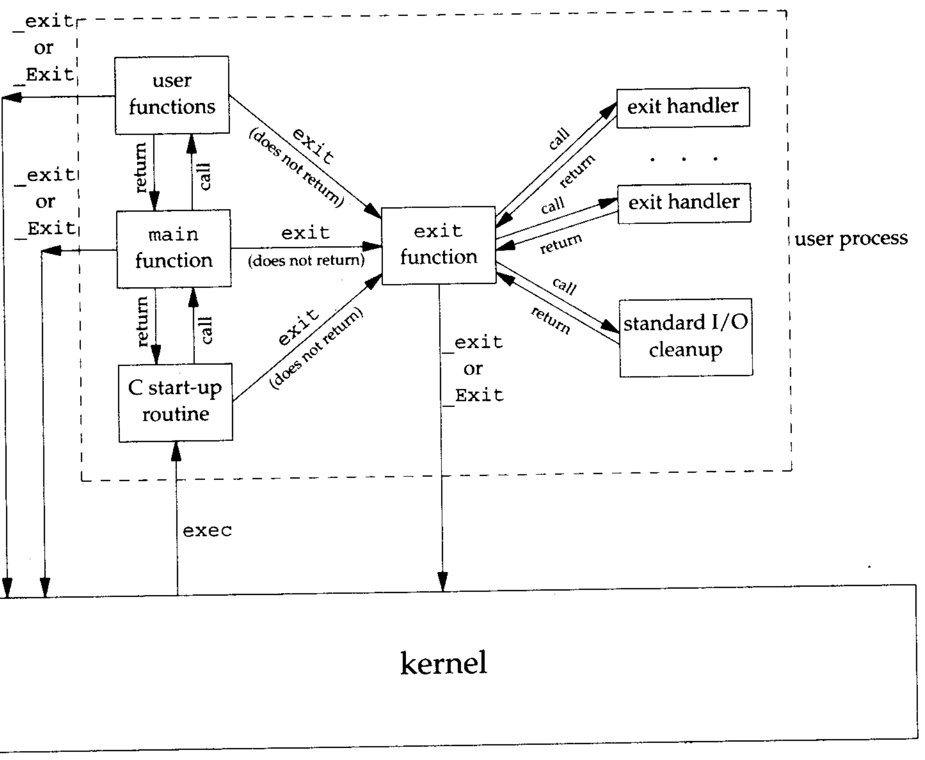

Figure 7.2 How a C program is started and how it terminates

Note that the only way a program is executed by the kernel is when one of the `exec` functions is called. The only way a process voluntarily terminates is when `_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\0 100

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

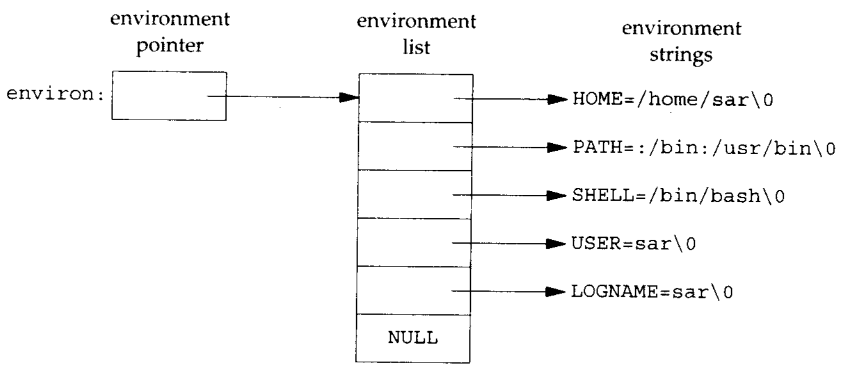

Figure 7.5 Environment consisting of five C character strings

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, 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 an 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. This is because the kernel sets it 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.

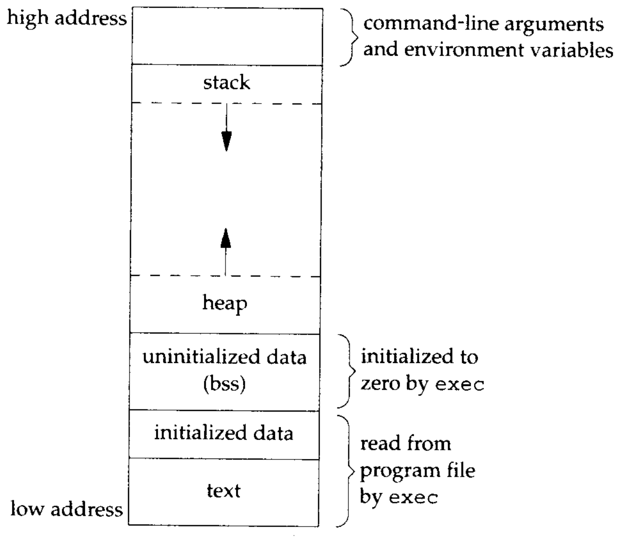
high address
low address
stack
heap
uninitialized data (bss)
initialized data
text
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

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
  79606    1536    916   82058   1408a  /usr/bin/cc
 619234   21120  18260  658614   a0cb6  /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. (This assumes 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:

```
$ cc -static hello1.c            prevent gcc from using shared libraries
$ ls -l a.out
-rwxrwxr-x  1 sar     475570 Feb 18 23:17 a.out
$ size a.out
   text     data     bss      dec      hex   filename
 375657     3780    3220   382657    5d6c1   a.out
```

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

```
$ cc hello1.c                    gcc defaults to use shared libraries
$ ls -l a.out
-rwxrwxr-x  1 sar      11410 Feb 18 23:19 a.out
$ size a.out
   text     data     bss      dec      hex   filename
    872      256       4     1132      46c   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 function `free` causes the space pointed to by *ptr* to be deallocated. This freed (4) 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 increase or decrease the size of a previously allocated area. (The most common usage is to increase an area.) For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call `realloc`. If there is room beyond the end of the (3) existing region for the requested space, then `realloc` doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, `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 shows the use of `realloc` with `getcwd` to handle any length pathname. Figure 17.36 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 `freed` 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 (8) 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; that space is kept in the `malloc` pool.

It is important to realize that most implementations allocate a little more space than (1) is requested and use the additional space for record keeping—the size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record-keeping information in a later block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping by writing before the start of the allocated area.

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.

6 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 continually increases; this is called leakage.) By not calling `free` to return unused space, the size of a process's address space slowly increases 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 alternate 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 also 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 size of the buffers. Free implementations of `malloc` and `free` based on quick-fit are readily available from several FTP sites.

`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, and others are 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 extensions are supported. Figure 7.7 lists the environment variables defined by the Single UNIX Specification and also notes which implementations support the variables. Any environment variable defined by POSIX.1 is marked with •; otherwise, it is an XSI extension. 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 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	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`	XSI	•	•	•	•	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 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`getenv`	•	•	•	•	•	•
`putenv`		XSI	•	•	•	•
`setenv`		•	•	•	•	
`unsetenv`		•	•	•	•	
`clearenv`				•		

Figure 7.8 Support for various environment list functions

`clearenv` 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);

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

int unsetenv(const char *name);

                                   All return: 0 if OK, nonzero 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; (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, on Linux and Solaris, the `putenv` implementation places the address of the string we pass to it directly into the environment list. In this case, it would be an error to pass it 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 simply 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

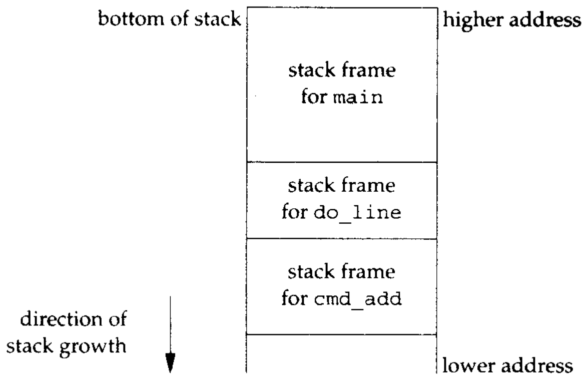

Figure 7.10 Stack frames after cmd_add has been called

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, 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, in the `cmd_add` function, we're only two levels down from `main`, but it's not uncommon to be five or more levels down from where we want to return to.) 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 is because 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 reason for the second argument is to allow us to have 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).

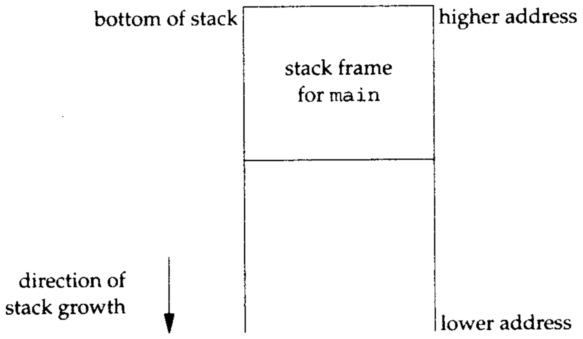

Figure 7.12 Stack frame after longjmp has been called

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 main is returned to by the longjmp, do these variables have values corresponding to 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 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:

```
$ cc 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
$ cc -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 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. There are numerous warnings about this 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>

#define DATAFILE    "datafile"

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, nonzero on error
```

These two functions are defined as XSI extensions 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 14.9).
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_LOCKS	The maximum number of file locks a process can hold. (This number also includes file leases, a Linux-specific feature. See the Linux fcntl(2) manual page for more information.)
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using mlock(2).
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 2.16 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_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_STACK	The maximum size in bytes of the stack. See Figure 7.6.
RLIMIT_VMEM	This is a synonym for RLIMIT_AS.

Limit	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
RLIMIT_AS	•		•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_LOCKS			•		
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_STACK	•	•	•	•	•
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 also that we must use a different `printf` format on platforms that define `rlim_t` to be an `unsigned long long` instead of an `unsigned long`.

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld  "
#else
#define FMT "%10ld  "
#endif
#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_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

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

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
    else
        printf(FMT, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
    else
        printf(FMT, limit.rlim_max);
    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:

```
$ ./a.out
RLIMIT_CORE      (infinite)  (infinite)
RLIMIT_CPU       (infinite)  (infinite)
RLIMIT_DATA       536870912   536870912
RLIMIT_FSIZE     (infinite)  (infinite)
RLIMIT_MEMLOCK   (infinite)  (infinite)
RLIMIT_NOFILE          1735        1735
RLIMIT_NPROC            867         867
RLIMIT_RSS       (infinite)  (infinite)
RLIMIT_SBSIZE    (infinite)  (infinite)
RLIMIT_STACK       67108864    67108864
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 in 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 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 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 both FreeBSD and 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 `printf`s 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 (475570 and 11410) don't equal the sums of their respective text and data sizes. Why?

7.9 In Section 7.7, why is there such a difference in the size of the executable file when using 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     *ptr;

    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.

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 share the text segment (Section 7.6).

Current 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 2.4.22 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 5.2.1 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 9 provides two threads libraries: one for POSIX threads (pthreads) and one for Solaris threads. The behavior of fork differs between the two thread libraries. For POSIX threads, fork creates a process containing only the calling thread, but for Solaris threads, fork creates a process containing copies of all threads from the process of the calling thread. To provide similar semantics as POSIX threads, Solaris provides the fork1 function, which can be used to create a process that duplicates only the calling thread, regardless of the thread library used. 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     glob = 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 */
        glob++;                 /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, 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. This depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize, 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 this 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 `printf` line, because the standard output buffer is flushed by the newline. But when we redirect standard output to a file, 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 3.8).

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 `forks` a child, then `waits` 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 `waiting` 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.

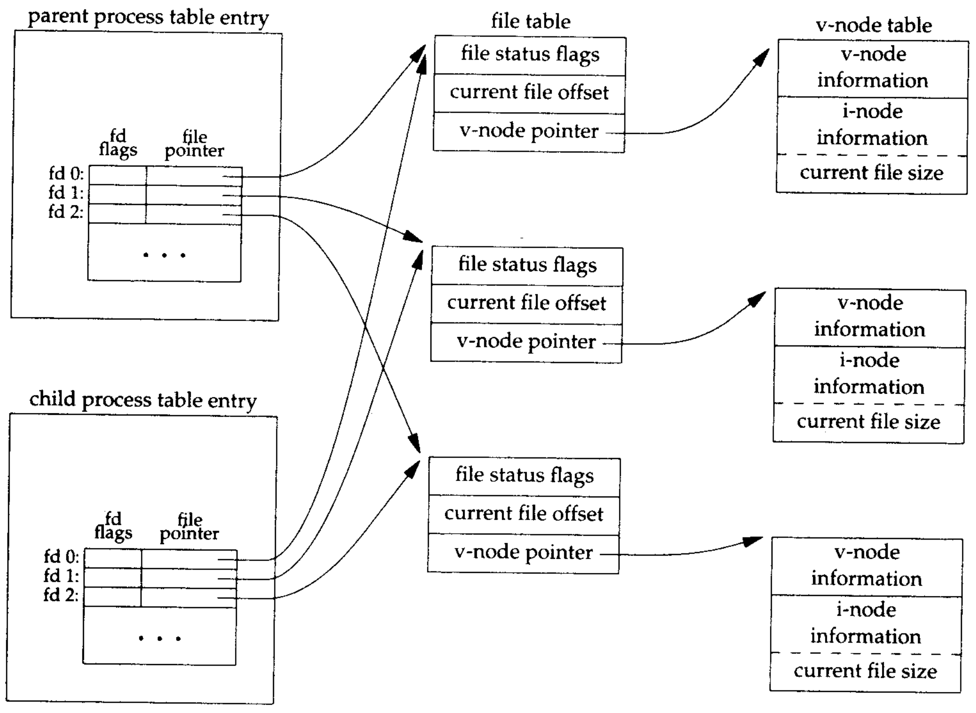

Figure 8.2 Sharing of open files between parent and child after `fork`

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 the case with network servers.

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

- Real user ID, real group ID, effective user ID, 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 value from `fork`
- 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
- 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 2.10 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 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 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 is marked as an obsolete interface in Version 3 of the Single UNIX Specification.

The vfork function is intended to create a new process when the purpose of the new process is to exec a new program (step 2 at the end of the previous section). The bare-bones shell in the program from Figure 1.7 is also an example of this type of program. 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, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System. (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     glob = 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 */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, 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 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 parent's printf 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 with 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` will 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 or not 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—for example, 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 the like.

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. 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 were 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 to indicate 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 extension to POSIX.1; `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 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9 all support the WCOREDUMP macro.

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)
```

Unfortunately, there is no portable way to map the signal numbers from WTERMSIG into descriptive names. (See Section 10.21 for one method.) We have to look at the <signal.h> header to verify that SIGABRT has a value of 6 and that SIGFPE has a value of 8. □

As we mentioned, if we have more than one child, wait returns on termination of any of the children. 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 is either 0 or is constructed from the bitwise OR of the constants in Figure 8.7.

Solaris supports 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 extension to POSIX.1).
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 forks 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 = %d\n", 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 XSI extension of 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`

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 Solaris provides support for `waitid`.

8.8 wait3 and wait4 Functions

Most UNIX system implementations provide two additional functions: `wait3` and `wait4`. Historically, these two variants descend from the BSD branch of the UNIX System. The only feature provided by these two functions that isn't provided by the `wait`, `waitid`, and `waitpid` functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);

                                Both return: process ID if OK, 0, or -1 on error
```

The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received, and the like. Refer to the `getrusage(2)` manual page for additional details. (This resource information differs from the resource limits we described in Section 7.11.) Figure 8.11 details the various arguments supported by the `wait` functions.

Function	*pid*	*options*	*rusage*	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`wait`				•	•	•	•	•
`waitid`	•	•		XSI				•
`waitpid`	•	•		•	•	•	•	•
`wait3`		•	•		•	•	•	•
`wait4`	•	•	•		•	•	•	•

Figure 8.11 Arguments supported by `wait` functions on various systems

The `wait3` function was included in earlier versions of the Single UNIX Specification. In Version 2, `wait3` was moved to the legacy category; `wait3` was removed from the specification in Version 3.

8.9 Race Conditions

For our purposes, a race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The `fork` function is a lively breeding ground for race conditions, if any of the logic after the `fork` either explicitly or implicitly depends on whether the parent or child runs first after the `fork`. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

We saw a potential race condition in the program in Figure 8.8 when the second child printed its parent process ID. If the second child runs before the first child, then its parent process will be the first child. But if the first child runs first and has enough time to `exit`, then the parent process of the second child is `init`. Even calling `sleep`, as we did, guarantees nothing. If the system was heavily loaded, the second child could resume after `sleep` returns, before the first child has a chance to run. Problems of this form can be difficult to debug because they tend to work "most of the time."

A process that wants to wait for a child to terminate must call one of the `wait` functions. If a process wants to wait for its parent to terminate, as in the program from Figure 8.8, a loop of the following form could be used:

```
while (getppid() != 1)
    sleep(1);
```

The problem with this type of loop, called *polling*, is that it wastes CPU time, as the caller is awakened every second to test the condition.

To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used, and we describe one way to do this in Section 10.16. Various forms of interprocess communication (IPC) can also be used. We'll discuss some of these in Chapters 15 and 17.

For a parent and child relationship, we often have the following scenario. After the `fork`, both the parent and the child have something to do. For example, the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent. In this example, we require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own. The following code illustrates this scenario:

```
#include  "apue.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

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

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */
WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

We assume that the header apue.h defines whatever variables are required. The five routines TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT, and WAIT_CHILD can be either macros or functions.

We'll show various ways to implement these TELL and WAIT routines in later chapters: Section 10.16 shows an implementation using signals; Figure 15.7 shows an implementation using pipes. Let's look at an example that uses these five routines.

Example

The program in Figure 8.12 outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Figure 8.12 Program with a race condition

We set the standard output unbuffered, so every character output generates a write. The goal in this example is to allow the kernel to switch between the two processes as often as possible to demonstrate the race condition. (If we didn't do this, we might never see the type of output that follows. Not seeing the erroneous output doesn't mean that the race condition doesn't exist; it simply means that we can't see it on this particular system.) The following actual output shows how the results can vary:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

We need to change the program in Figure 8.12 to use the TELL and WAIT functions. The program in Figure 8.13 does this. The lines preceded by a plus sign are new lines.

```
  #include "apue.h"

  static void charatatime(char *);

  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }

  static void
  charatatime(char *str)
  {
      char    *ptr;
      int     c;

      setbuf(stdout, NULL);           /* set unbuffered */
      for (ptr = str; (c = *ptr++) != 0; )
          putc(c, stdout);
  }
```

Figure 8.13 Modification of Figure 8.12 to avoid race condition

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

In the program shown in Figure 8.13, the parent goes first. The child goes first if we change the lines following the fork to be

```
} else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();          /* child goes first */
    charatatime("output from parent\n");
}
```

Exercise 8.3 continues this example. □

8.10 exec Functions

We mentioned in Section 8.3 that one use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process—its text, data, heap, and stack segments—with a brand new program from disk.

There are six different exec functions, but we'll often simply refer to "the exec function," which means that we could use any of the six functions. These six functions round out the UNIX System process control primitives. With fork, we can create new processes; and with the exec functions, we can initiate new programs. The exit function and the wait functions handle termination and waiting for termination. These are the only process control primitives we need. We'll use these primitives in later sections to build additional functions, such as popen and system.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

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

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

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

All six return: –1 on error, no return on success

The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a *filename* argument is specified

- If *filename* contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.

The PATH variable contains a list of directories, called path prefixes, that are separated by colons. For example, the *name=value* environment string

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

specifies four directories to search. The last path prefix specifies the current directory. (A zero-length prefix also means the current directory. It can be specified as a colon at the beginning of the *value*, two colons in a row, or a colon at the end of the *value*.)

> There are security reasons for *never* including the current directory in the search path. See Garfinkel et al. [2003].

If either execlp or execvp finds an executable file using one of the path prefixes, but the file isn't a machine executable that was generated by the link editor, the function assumes that the file is a shell script and tries to invoke /bin/sh with the *filename* as input to the shell.

The next difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

Before using ISO C prototypes, the normal way to show the command-line arguments for the three functions execl, execle, and execlp was

```
char *arg0, char *arg1, ..., char *argn, (char *)0
```

This specifically shows that the final command-line argument is followed by a null pointer. If this null pointer is specified by the constant 0, we must explicitly cast it to a pointer; if we don't, it's interpreted as an integer argument. If the size of an integer is different from the size of a char *, the actual arguments to the exec function will be wrong.

The final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program. (Recall our discussion of the environment strings in Section 7.9 and Figure 7.8. We mentioned that if the system supported such functions as setenv and putenv, we could change the current environment and the environment of any subsequent child processes, but we couldn't affect the environment of the parent process.) Normally, a process allows its environment to be propagated to its children, but in some cases, a process wants to specify a certain environment for a child. One example of the latter is the `login` program when a new login shell is initiated. Normally, `login` creates a specific environment with only a few variables defined and lets us, through the shell start-up file, add variables to the environment when we log in.

Before using ISO C prototypes, the arguments to `execle` were shown as

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

This specifically shows that the final argument is the address of the array of character pointers to the environment strings. The ISO C prototype doesn't show this, as all the command-line arguments, the null pointer, and the *envp* pointer are shown with the ellipsis notation (...).

The arguments for these six `exec` functions are difficult to remember. The letters in the function names help somewhat. The letter `p` means that the function takes a *filename* argument and uses the `PATH` environment variable to find the executable file. The letter `l` means that the function takes a list of arguments and is mutually exclusive with the letter `v`, which means that it takes an *argv*`[]` vector. Finally, the letter `e` means that the function takes an *envp*`[]` array instead of using the current environment. Figure 8.14 shows the differences among these six functions.

Function	*pathname*	*filename*	Arg list	*argv*`[]`	`environ`	*envp*`[]`
`execl`	•		•		•	
`execlp`		•	•		•	
`execle`	•		•			•
`execv`	•			•	•	
`execvp`		•		•	•	
`execve`	•			•		•
(letter in name)		`p`	`l`	`v`		`e`

Figure 8.14 Differences among the six `exec` functions

Every system has a limit on the total size of the argument list and the environment list. From Section 2.5.2 and Figure 2.8, this limit is given by `ARG_MAX`. This value must be at least 4,096 bytes on a POSIX.1 system. We sometimes encounter this limit when using the shell's filename expansion feature to generate a list of filenames. On some systems, for example, the command

```
grep getrlimit /usr/share/man/*/*
```

can generate a shell error of the form

```
Argument list too long
```

Historically, the limit in older System V implementations was 5,120 bytes. Older BSD systems had a limit of 20,480 bytes. The limit in current systems is much higher. (See the output from the program in Figure 2.13, which is summarized in Figure 2.14.)

To get around the limitation in argument list size, we can use the xargs(1) command to break up long argument lists. To look for all the occurrences of getrlimit in the man pages on our system, we could use

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

If the man pages on our system are compressed, however, we could try

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

We use the type -f option to the find command to restrict the list to contain only regular files, because the grep commands can't search for patterns in directories, and we want to avoid unnecessary error messages.

We've mentioned that the process ID does not change after an exec, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Values for tms_utime, tms_stime, tms_cutime, and tms_cstime

The handling of open files depends on the value of the close-on-exec flag for each descriptor. Recall from Figure 3.6 and our mention of the FD_CLOEXEC flag in Section 3.14 that every open descriptor in a process has a close-on-exec flag. If this flag is set, the descriptor is closed across an exec. Otherwise, the descriptor is left open across the exec. The default is to leave the descriptor open across the exec unless we specifically set the close-on-exec flag using fcntl.

POSIX.1 specifically requires that open directory streams (recall the opendir function from Section 4.21) be closed across an exec. This is normally done by the opendir function calling fcntl to set the close-on-exec flag for the descriptor corresponding to the open directory stream.

Note that the real user ID and the real group ID remain the same across the exec, but the effective IDs can change, depending on the status of the set-user-ID and the set-group-ID bits for the program file that is executed. If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file. Otherwise, the effective user ID is not changed (it's not set to the real user ID). The group ID is handled in the same way.

In many UNIX system implementations, only one of these six functions, execve, is a system call within the kernel. The other five are just library functions that eventually invoke this system call. We can illustrate the relationship among these six functions as shown in Figure 8.15.

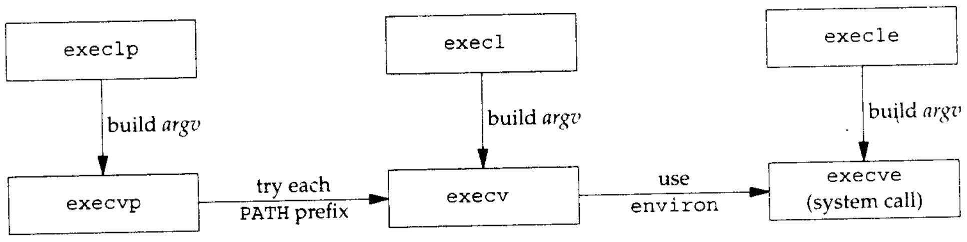

Figure 8.15 Relationship of the six exec functions

In this arrangement, the library functions execlp and execvp process the PATH environment variable, looking for the first path prefix that contains an executable file named *filename*.

Example

The program in Figure 8.16 demonstrates the exec functions.

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Figure 8.16 Example of exec functions

We first call `execle`, which requires a pathname and a specific environment. The next call is to `execlp`, which uses a filename and passes the caller's environment to the new program. The only reason the call to `execlp` works is that the directory `/home/sar/bin` is one of the current path prefixes. Note also that we set the first argument, `argv[0]` in the new program, to be the filename component of the pathname. Some shells set this argument to be the complete pathname. This is a convention only. We can set `argv[0]` to any string we like. The `login` command does this when it executes the shell. Before executing the shell, `login` adds a dash as a prefix to `argv[0]` to indicate to the shell that it is being invoked as a login shell. A login shell will execute the start-up profile commands, whereas a nonlogin shell will not.

The program `echoall` that is executed twice in the program in Figure 8.16 is shown in Figure 8.17. It is a trivial program that echoes all its command-line arguments and its entire environment list.

```
#include "apue.h"

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

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

    for (ptr = environ; *ptr != 0; ptr++)    /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Figure 8.17 Echo all command-line arguments and all environment strings

When we execute the program from Figure 8.16, we get

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                                    47 more lines that aren't shown
HOME=/home/sar
```

Note that the shell prompt appeared before the printing of `argv[0]` from the second `exec`. This is because the parent did not `wait` for this child process to finish. □

8.11 Changing User IDs and Group IDs

In the UNIX System, privileges, such as being able to change the system's notion of the current date, and access control, such as being able to read or write a particular file, are based on user and group IDs. When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

In general, we try to use the *least-privilege* model when we design our applications. Following this model, our programs should use the least privilege necessary to accomplish any given task. This reduces the likelihood that security can be compromised by a malicious user trying to trick our programs into using their privileges in unintended ways.

We can set the real user ID and effective user ID with the `setuid` function. Similarly, we can set the real group ID and the effective group ID with the `setgid` function.

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);

                                        Both return: 0 if OK, -1 on error
```

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

1. If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to *uid*.
2. If the process does not have superuser privileges, but *uid* equals either the real user ID or the saved set-user-ID, setuid sets only the effective user ID to *uid*. The real user ID and the saved set-user-ID are not changed.
3. If neither of these two conditions is true, errno is set to EPERM, and −1 is returned.

Here, we are assuming that _POSIX_SAVED_IDS is true. If this feature isn't provided, then delete all preceding references to the saved set-user-ID.

The saved IDs are a mandatory feature in the 2001 version of POSIX.1. They used to be optional in older versions of POSIX. To see whether an implementation supports this feature, an application can test for the constant _POSIX_SAVED_IDS at compile time or call sysconf with the _SC_SAVED_IDS argument at runtime.

We can make a few statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally, the real user ID is set by the login(1) program when we log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid.
2. The effective user ID is set by the exec functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the exec functions leave the effective user ID as its current value. We can call setuid at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
3. The saved set-user-ID is copied from the effective user ID by exec. If the file's set-user-ID bit is set, this copy is saved after exec stores the effective user ID from the file's user ID.

Figure 8.18 summarizes the various ways these three user IDs can be changed.

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Figure 8.18 Ways to change the three user IDs

Note that we can obtain only the current value of the real user ID and the effective user ID with the functions getuid and geteuid from Section 8.2. We can't obtain the current value of the saved set-user-ID.

Example

To see the utility of the saved set-user-ID feature, let's examine the operation of a program that uses it. We'll look at the man(1) program, which is used to display online manual pages. The man program can be installed either set-user-ID or set-group-ID to a specific user or group, usually one reserved for man itself. The man program can be made to read and possibly overwrite files in locations that are chosen either through a configuration file (usually `/etc/man.config` or `/etc/manpath.config`) or using a command-line option.

The man program might have to execute several other commands to process the files containing the manual page to be displayed. To prevent being tricked into running the wrong commands or overwriting the wrong files, the man command has to switch between two sets of privileges: those of the user running the man command and those of the user that owns the man executable file. The following steps take place.

1. Assuming that the man program file is owned by the user name man and has its set-user-ID bit set, when we exec it, we have

   real user ID = our user ID
   effective user ID = man
   saved set-user-ID = man

2. The man program accesses the required configuration files and manual pages. These files are owned by the user name man, but because the effective user ID is man, file access is allowed.

3. Before man runs any command on our behalf, it calls `setuid(getuid())`. Because we are not a superuser process, this changes only the effective user ID. We have

   real user ID = our user ID (unchanged)
   effective user ID = our user ID
   saved set-user-ID = man (unchanged)

   Now the man process is running with our user ID as its effective user ID. This means that we can access only the files to which we have normal access. We have no additional permissions. It can safely execute any filter on our behalf.

4. When the filter is done, man calls `setuid(`*euid*`)`, where *euid* is the numerical user ID for the user name man. (This was saved by man by calling `geteuid`.) This call is allowed because the argument to `setuid` equals the saved set-user-ID. (This is why we need the saved set-user-ID.) Now we have

   real user ID = our user ID (unchanged)
   effective user ID = man
   saved set-user-ID = man (unchanged)

5. The man program can now operate on its files, as its effective user ID is man.

By using the saved set-user-ID in this fashion, we can use the extra privileges granted to us by the set-user-ID of the program file at the beginning of the process and at the end of the process. In between, however, the process runs with our normal permissions. If we weren't able to switch back to the saved set-user-ID at the end, we might be tempted to retain the extra permissions the whole time we were running (which is asking for trouble).

Let's look at what happens if man spawns a shell for us while it is running. (The shell is spawned using `fork` and `exec`.) Because the real user ID and the effective user ID are both our normal user ID (step 3), the shell has no extra permissions. The shell can't access the saved set-user-ID that is set to man while man is running, because the saved set-user-ID for the shell is copied from the effective user ID by `exec`. So in the child process that does the `exec`, all three user IDs are our normal user ID.

Our description of how man uses the `setuid` function is not correct if the program is set-user-ID to root, because a call to `setuid` with superuser privileges sets all three user IDs. For the example to work as described, we need `setuid` to set only the effective user ID. □

`setreuid` and `setregid` Functions

Historically, BSD supported the swapping of the real user ID and the effective user ID with the `setreuid` function.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);

                                   Both return: 0 if OK, -1 on error
```

We can supply a value of –1 for any of the arguments to indicate that the corresponding ID should remain unchanged.

The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user-ID operations. When the saved set-user-ID feature was introduced with POSIX.1, the rule was enhanced to also allow an unprivileged user to set its effective user ID to its saved set-user-ID.

Both `setreuid` and `setregid` are XSI extensions in the Single UNIX Specification. As such, all UNIX System implementations are expected to provide support for them.

4.3BSD didn't have the saved set-user-ID feature described earlier. It used `setreuid` and `setregid` instead. This allowed an unprivileged user to swap back and forth between the two values. Be aware, however, that when programs that used this feature spawned a shell, they had to set the real user ID to the normal user ID before the `exec`. If they didn't do this, the real user ID could be privileged (from the swap done by `setreuid`) and the shell process could call `setreuid` to swap the two and assume the permissions of the more privileged user. As a defensive programming measure to solve this problem, programs set both the real user ID and the effective user ID to the normal user ID before the call to `exec` in the child.

`seteuid` and `setegid` Functions

POSIX.1 includes the two functions `seteuid` and `setegid`. These functions are similar to `setuid` and `setgid`, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);

                                        Both return: 0 if OK, -1 on error
```

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to *uid*. (This differs from the `setuid` function, which changes all three user IDs.)

Figure 8.19 summarizes all the functions that we've described here that modify the three user IDs.

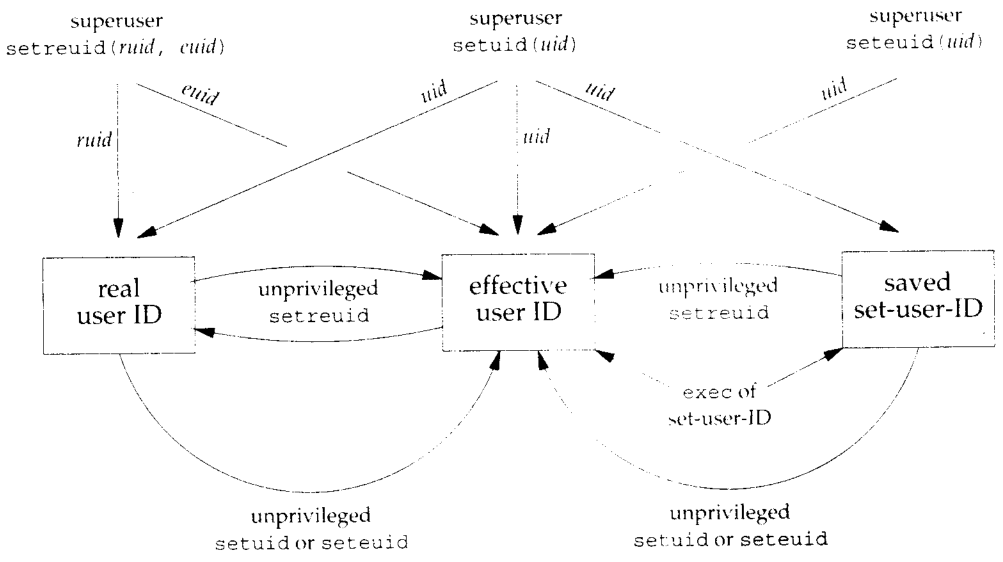

Figure 8.19 Summary of all the functions that set the various user IDs

Group IDs

Everything that we've said so far in this section also applies in a similar fashion to group IDs. The supplementary group IDs are not affected by `setgid`, `setregid`, or `setegid`.

8.12 Interpreter Files

All contemporary UNIX systems support interpreter files. These files are text files that begin with a line of the form

```
#! pathname [ optional-argument ]
```

The space between the exclamation point and the *pathname* is optional. The most common of these interpreter files begin with the line

```
#!/bin/sh
```

The *pathname* is normally an absolute pathname, since no special operations are performed on it (i.e., `PATH` is not used). The recognition of these files is done within the kernel as part of processing the `exec` system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the *pathname* on the first line of the interpreter file. Be sure to differentiate between the interpreter file—a text file that begins with `#!`—and the interpreter, which is specified by the *pathname* on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the `#!`, the *pathname*, the optional argument, the terminating newline, and any spaces.

On FreeBSD 5.2.1, this limit is 128 bytes. Mac OS X 10.3 extends this limit to 512 bytes. Linux 2.4.22 supports a limit of 127 bytes, whereas Solaris 9 places the limit at 1,023 bytes.

Example

Let's look at an example to see what the kernel does with the arguments to the `exec` function when the file being executed is an interpreter file and the optional argument on the first line of the interpreter file. The program in Figure 8.20 execs an interpreter file.

```
#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) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0)  /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Figure 8.20 A program that execs an interpreter file

The following shows the contents of the one-line interpreter file that is executed and the result from running the program in Figure 8.20:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

The program `echoarg` (the interpreter) just echoes each of its command-line arguments. (This is the program from Figure 7.4.) Note that when the kernel `exec`s the interpreter (`/home/sar/bin/echoarg`), `argv[0]` is the *pathname* of the interpreter, `argv[1]` is the optional argument from the interpreter file, and the remaining arguments are the *pathname* (`/home/sar/bin/testinterp`) and the second and third arguments from the call to `execl` in the program shown in Figure 8.20 (`myarg1` and `MY ARG2`). Both `argv[1]` and `argv[2]` from the call to `execl` have been shifted right two positions. Note that the kernel takes the *pathname* from the `execl` call instead of the first argument (`testinterp`), on the assumption that the *pathname* might contain more information than the first argument. □

Example

A common use for the optional argument following the interpreter *pathname* is to specify the `-f` option for programs that support this option. For example, an `awk(1)` program can be executed as

```
awk -f myfile
```

which tells `awk` to read the `awk` program from the file `myfile`.

Systems derived from UNIX System V often include two versions of the `awk` language. On these systems, `awk` is often called "old `awk`" and corresponds to the original version distributed with Version 7. In contrast, `nawk` (new `awk`) contains numerous enhancements and corresponds to the language described in Aho, Kernighan, and Weinberger [1988]. This newer version provides access to the command-line arguments, which we need for the example that follows. Solaris 9 provides both versions.

The `awk` program is one of the utilities included by POSIX in its 1003.2 standard, which is now part of the base POSIX.1 specification in the Single UNIX Specification. This utility is also based on the language described in Aho, Kernighan, and Weinberger [1988].

The version of `awk` in Mac OS X 10.3 is based on the Bell Laboratories version that Lucent has placed in the public domain. FreeBSD 5.2.1 and Linux 2.4.22 ship with GNU `awk`, called `gawk`, which is linked to the name `awk`. The `gawk` version conforms to the POSIX standard, but also includes other extensions. Because they are more up-to-date, the version of `awk` from Bell Laboratories and `gawk` are preferred to either `nawk` or old `awk`. (The version of `awk` from Bell Laboratories is available at `http://cm.bell-labs.com/cm/cs/awkbook/index.html`.)

Using the -f option with an interpreter file lets us write

```
#!/bin/awk -f
```
(awk program follows in the interpreter file)

For example, Figure 8.21 shows /usr/local/bin/awkexample (an interpreter file).

```
#!/bin/awk -f
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

Figure 8.21 An awk program as an interpreter file

If one of the path prefixes is /usr/local/bin, we can execute the program in Figure 8.21 (assuming that we've turned on the execute bit for the file) as

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

When /bin/awk is executed, its command-line arguments are

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

The pathname of the interpreter file (/usr/local/bin/awkexample) is passed to the interpreter. The filename portion of this pathname (what we typed to the shell) isn't adequate, because the interpreter (/bin/awk in this example) can't be expected to use the PATH variable to locate files. When it reads the interpreter file, awk ignores the first line, since the pound sign is awk's comment character.

We can verify these command-line arguments with the following commands:

```
$ /bin/su                                    become superuser
Password:                                    enter superuser password
# mv /bin/awk /bin/awk.save                  save the original program
# cp /home/sar/bin/echoarg /bin/awk          and replace it temporarily
# suspend                                    suspend the superuser shell using job control
[1] + Stopped          /bin/su
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                         resume superuser shell using job control
/bin/su
# mv /bin/awk.save /bin/awk                  restore the original program
# exit                                       and exit the superuser shell
```

In this example, the `-f` option for the interpreter is required. As we said, this tells `awk` where to look for the `awk` program. If we remove the `-f` option from the interpreter file, an error message usually results when we try to run it. The exact text of the message varies, depending on where the interpreter file is stored and whether the remaining arguments represent existing files. This is because the command-line arguments in this case are

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

and `awk` is trying to interpret the string `/usr/local/bin/awkexample` as an `awk` program. If we couldn't pass at least a single optional argument to the interpreter (`-f` in this case), these interpreter files would be usable only with the shells. □

Are interpreter files required? Not really. They provide an efficiency gain for the user at some expense in the kernel (since it's the kernel that recognizes these files). Interpreter files are useful for the following reasons.

1. They hide that certain programs are scripts in some other language. For example, to execute the program in Figure 8.21, we just say

   `awkexample` *optional-arguments*

   instead of needing to know that the program is really an `awk` script that we would otherwise have to execute as

   `awk -f awkexample` *optional-arguments*

2. Interpreter scripts provide an efficiency gain. Consider the previous example again. We could still hide that the program is an `awk` script, by wrapping it in a shell script:

   ```
   awk 'BEGIN {
       for (i = 0; i < ARGC; i++)
           printf "ARGV[%d] = %s\n", i, ARGV[i]
       exit
   }' $*
   ```

   The problem with this solution is that more work is required. First, the shell reads the command and tries to `execlp` the filename. Because the shell script is an executable file, but isn't a machine executable, an error is returned, and `execlp` assumes that the file is a shell script (which it is). Then `/bin/sh` is executed with the pathname of the shell script as its argument. The shell correctly runs our script, but to run the `awk` program, the shell does a `fork`, `exec`, and `wait`. Thus, there is more overhead in replacing an interpreter script with a shell script.

3. Interpreter scripts let us write shell scripts using shells other than `/bin/sh`. When it finds an executable file that isn't a machine executable, `execlp` has to choose a shell to invoke, and it always uses `/bin/sh`. Using an interpreter script, however, we can simply write

```
#!/bin/csh
```
(C shell script follows in the interpreter file)

Again, we could wrap this all in a /bin/sh script (that invokes the C shell), as we described earlier, but more overhead is required.

None of this would work as we've shown if the three shells and awk didn't use the pound sign as their comment character.

8.13 system Function

It is convenient to execute a command string from within a program. For example, assume that we want to put a time-and-date stamp into a certain file. We could use the functions we describe in Section 6.10 to do this: call time to get the current calendar time, then call localtime to convert it to a broken-down time, and then call strftime to format the result, and write the results to the file. It is much easier, however, to say

```
system("date > file");
```

ISO C defines the system function, but its operation is strongly system dependent. POSIX.1 includes the system interface, expanding on the ISO C definition to describe its behavior in a POSIX environment.

```
#include <stdlib.h>

int system(const char *cmdstring);

                                        Returns: (see below)
```

If *cmdstring* is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, system is always available.

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

1. If either the fork fails or waitpid returns an error other than EINTR, system returns −1 with errno set to indicate the error.
2. If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
3. Otherwise, all three functions—fork, exec, and waitpid—succeed, and the return value from system is the termination status of the shell, in the format specified for waitpid.

Some older implementations of system returned an error (EINTR) if waitpid was interrupted by a caught signal. Because there is no cleanup strategy that an application can use to recover from this type of error, POSIX later added the requirement that system not return an error in this case. (We discuss interrupted system calls in Section 10.5.)

Figure 8.22 shows an implementation of the system function. The one feature that it doesn't handle is signals. We'll update this function with signal handling in Section 10.18.

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

int
system(const char *cmdstring)   /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Figure 8.22 The system function, without signal handling

The shell's -c option tells it to take the next command-line argument—*cmdstring*, in this case—as its command input instead of reading from standard input or from a given file. The shell parses this null-terminated C string and breaks it up into separate command-line arguments for the command. The actual command string that is passed to the shell can contain any valid shell commands. For example, input and output redirection using < and > can be used.

If we didn't use the shell to execute the command, but tried to execute the command ourself, it would be more difficult. First, we would want to call execlp instead of execl, to use the PATH variable, like the shell. We would also have to break up the null-terminated C string into separate command-line arguments for the call to execlp. Finally, we wouldn't be able to use any of the shell metacharacters.

Note that we call _exit instead of exit. We do this to prevent any standard I/O buffers, which would have been copied from the parent to the child across the fork, from being flushed in the child.

We can test this version of `system` with the program shown in Figure 8.23. (The `pr_exit` function was defined in Figure 8.5.)

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

int
main(void)
{
    int     status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Figure 8.23 Calling the `system` function

Running the program in Figure 8.23 gives us

```
$ ./a.out
Sun Mar 21 18:41:32 EST 2004
normal termination, exit status = 0      for date
sh: nosuchcommand: command not found
normal termination, exit status = 127    for nosuchcommand
sar      :0       Mar 18 19:45
sar      pts/0    Mar 18 19:45 (:0)
sar      pts/1    Mar 18 19:45 (:0)
sar      pts/2    Mar 18 19:45 (:0)
sar      pts/3    Mar 18 19:45 (:0)
normal termination, exit status = 44     for exit
```

The advantage in using `system`, instead of using `fork` and `exec` directly, is that `system` does all the required error handling and (in our next version of this function in Section 10.18) all the required signal handling.

Earlier systems, including SVR3.2 and 4.3BSD, didn't have the `waitpid` function available. Instead, the parent waited for the child, using a statement such as

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

A problem occurs if the process that calls `system` has spawned its own children before calling `system`. Because the `while` statement above keeps looping until the child that was generated by `system` terminates, if any children of the process terminate before the process identified by pid, then the process ID and termination status of these other children are discarded by the while statement. Indeed, this inability to wait for a specific child is one of the reasons given in the POSIX.1 Rationale for including the waitpid function. We'll see in Section 15.3 that the same problem occurs with the popen and pclose functions, if the system doesn't provide a waitpid function.

Set-User-ID Programs

What happens if we call system from a set-user-ID program? Doing so is a security hole and should never be done. Figure 8.24 shows a simple program that just calls system for its command-line argument.

```
#include "apue.h"

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

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Figure 8.24 Execute the command-line argument using system

We'll compile this program into the executable file tsys.

Figure 8.25 shows another simple program that prints its real and effective user IDs.

```
#include "apue.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

Figure 8.25 Print real and effective user IDs

We'll compile this program into the executable file printuids. Running both programs gives us the following:

```
$ tsys printuids                         normal execution, no special privileges
real uid = 205, effective uid = 205
normal termination, exit status = 0
$ su                                     become superuser
Password:                                enter superuser password
# chown root tsys                        change owner
# chmod u+s tsys                         make set-user-ID
# ls -l tsys                             verify file's permissions and owner
-rwsrwxr-x  1 root     16361 Mar 16 16:59 tsys
# exit                                   leave superuser shell
$ tsys printuids
real uid = 205, effective uid = 0        oops, this is a security hole
normal termination, exit status = 0
```

The superuser permissions that we gave the `tsys` program are retained across the `fork` and `exec` that are done by `system`.

> When `/bin/sh` is `bash` version 2, the previous example doesn't work, because `bash` will reset the effective user ID to the real user ID when they don't match.

If it is running with special permissions—either set-user-ID or set-group-ID—and wants to spawn another process, a process should use `fork` and `exec` directly, being certain to change back to normal permissions after the `fork`, before calling `exec`. The `system` function should *never* be used from a set-user-ID or a set-group-ID program.

> One reason for this admonition is that `system` invokes the shell to parse the command string, and the shell uses its `IFS` variable as the input field separator. Older versions of the shell didn't reset this variable to a normal set of characters when invoked. This allowed a malicious user to set `IFS` before `system` was called, causing `system` to execute a different program.

8.14 Process Accounting

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates. These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on. We'll take a closer look at these accounting records in this section, as it gives us a chance to look at processes again and to use the `fread` function from Section 5.9.

> Process accounting is not specified by any of the standards. Thus, all the implementations have annoying differences. For example, the I/O counts maintained on Solaris 9 are in units of bytes, whereas FreeBSD 5.2.1 and Mac OS X 10.3 maintain units of blocks, although there is no distinction between different block sizes, making the counter effectively useless. Linux 2.4.22, on the other hand, doesn't try to maintain I/O statistics at all.
>
> Each implementation also has its own set of administrative commands to process raw accounting data. For example, Solaris provides `runacct(1m)` and `acctcom(1)`, whereas FreeBSD provides the `sa(8)` command to process and summarize the raw accounting data.

A function we haven't described (`acct`) enables and disables process accounting. The only use of this function is from the `accton(8)` command (which happens to be one of the few similarities among platforms). A superuser executes accton with a pathname argument to enable accounting. The accounting records are written to the specified file, which is usually /var/account/acct on FreeBSD and Mac OS X, /var/account/pacct on Linux, and /var/adm/pacct on Solaris. Accounting is turned off by executing accton without any arguments.

The structure of the accounting records is defined in the header <sys/acct.h> and looks something like

```
typedef  u_short comp_t;   /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char    ac_flag;       /* flag (see Figure 8.26) */
  char    ac_stat;       /* termination status (signal & core flag only) */
                         /* (Solaris only) */
  uid_t   ac_uid;        /* real user ID */
  gid_t   ac_gid;        /* real group ID */
  dev_t   ac_tty;        /* controlling terminal */
  time_t  ac_btime;      /* starting calendar time */
  comp_t  ac_utime;      /* user CPU time (clock ticks) */
  comp_t  ac_stime;      /* system CPU time (clock ticks) */
  comp_t  ac_etime;      /* elapsed time (clock ticks) */
  comp_t  ac_mem;        /* average memory usage */
  comp_t  ac_io;         /* bytes transferred (by read and write) */
                         /* "blocks" on BSD systems */
  comp_t  ac_rw;         /* blocks read or written */
                         /* (not present on BSD systems) */
  char    ac_comm[8];    /* command name: [8] for Solaris, */
                         /* [10] for Mac OS X, [16] for FreeBSD, and */
                         /* [17] for Linux */
};
```

The ac_flag member records certain events during the execution of the process. These events are described in Figure 8.26.

ac_flag	Description	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
AFORK	process is the result of fork, but never called exec	•	•	•	•
ASU	process used superuser privileges		•	•	•
ACOMPAT	process used compatibility mode				
ACORE	process dumped core	•	•	•	
AXSIG	process was killed by a signal	•	•	•	
AEXPND	expanded accounting entry				•

Figure 8.26 Values for ac_flag from accounting record

The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork. Each accounting record is written when the process terminates. This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started. To know the starting order, we would have to go through the accounting file and sort by the starting calendar time. But this isn't perfect, since calendar times are in units of seconds (Section 1.10), and it's possible for many processes to be started in any given second. Alternatively, the elapsed time is given in clock ticks, which are usually between 60 and 128 ticks per second. But we don't know the ending time of a process; all we know is its starting time and ending order. This means that even though the elapsed time is more accurate than the starting time, we still can't reconstruct the exact starting order of various processes, given the data in the accounting file.

The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a `fork`, not when a new program is executed. Although `exec` doesn't create a new accounting record, the command name changes, and the AFORK flag is cleared. This means that if we have a chain of three programs—A `execs` B, then B `execs` C, and C `exits`—only a single accounting record is written. The command name in the record corresponds to program C, but the CPU times, for example, are the sum for programs A, B, and C.

Example

To have some accounting data to examine, we'll create a test program to implement the diagram shown in Figure 8.27.

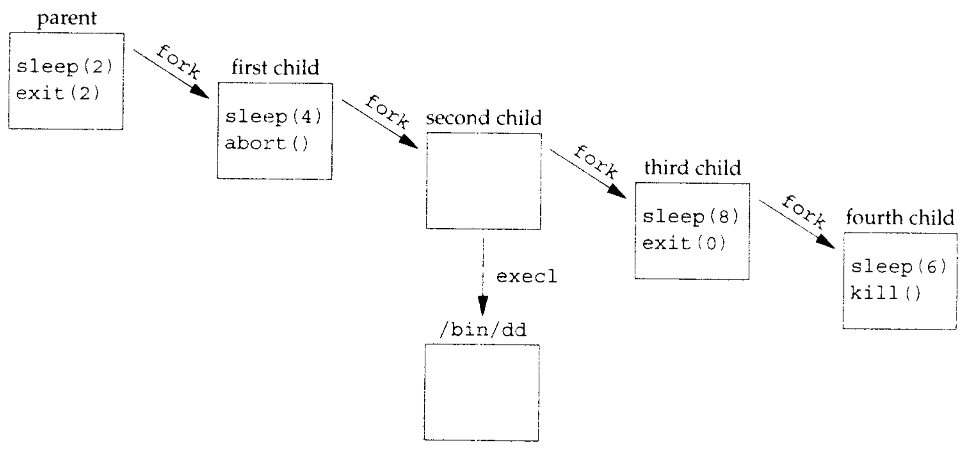

Figure 8.27 Process structure for accounting example

The source for the test program is shown in Figure 8.28. It calls `fork` four times. Each child does something different and then terminates.

```
#include "apue.h"

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

                                /* first child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                /* terminate with core dump */
    }

                                /* second child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
        exit(7);                /* shouldn't get here */
    }

                                /* third child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                /* normal exit */
    }

                                /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL);    /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```

Figure 8.28 Program to generate accounting data

We'll run the test program on Solaris and then use the program in Figure 8.29 to print out selected fields from the accounting records.

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

#ifdef HAS_SA_STAT
#define FMT "%-*.*s  e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n"
#else
#define FMT "%-*.*s  e = %6ld, chars = %7ld, %c %c %c %c\n"
#endif
#ifndef HAS_ACORE
#define ACORE 0
#endif
#ifndef HAS_AXSIG
#define AXSIG 0
#endif

static unsigned long
compt2ulong(comp_t comptime)    /* convert comp_t to unsigned long */
{
    unsigned long   val;
    int             exp;

    val = comptime & 0x1fff;    /* 13-bit fraction */
    exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
int
main(int argc, char *argv[])
{
    struct acct     acdata;
    FILE            *fp;

    if (argc != 2)
        err_quit("usage: pracct filename");
    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf(FMT, (int)sizeof(acdata.ac_comm),
            (int)sizeof(acdata.ac_comm), acdata.ac_comm,
            compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
#ifdef HAS_SA_STAT
            (unsigned char) acdata.ac_stat,
#endif
            acdata.ac_flag & ACORE ? 'D' : ' ',
            acdata.ac_flag & AXSIG ? 'X' : ' ',
            acdata.ac_flag & AFORK ? 'F' : ' ',
            acdata.ac_flag & ASU   ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}
```

Figure 8.29 Print selected fields from system's accounting file

BSD-derived platforms don't support the `ac_flag` member, so we define the `HAS_SA_STAT` constant on the platforms that do support this member. Basing the defined symbol on the feature instead of on the platform reads better and allows us to modify the program simply by adding the additional definition to our compilation command. The alternative would be to use

```
#if defined(BSD) || defined(MACOS)
```

which becomes unwieldy as we port our application to additional platforms.

We define similar constants to determine whether the platform supports the `ACORE` and `AXSIG` accounting flags. We can't use the flag symbols themselves, because on Linux, they are defined as `enum` values, which we can't use in a `#ifdef` expression.

To perform our test, we do the following:

1. Become superuser and enable accounting, with the `accton` command. Note that when this command terminates, accounting should be on; therefore, the first record in the accounting file should be from this command.
2. Exit the superuser shell and run the program in Figure 8.28. This should append six records to the accounting file: one for the superuser shell, one for the test parent, and one for each of the four test children.

   A new process is not created by the `execl` in the second child. There is only a single accounting record for the second child.
3. Become superuser and turn accounting off. Since accounting is off when this `accton` command terminates, it should not appear in the accounting file.
4. Run the program in Figure 8.29 to print the selected fields from the accounting file.

The output from step 4 follows. We have appended to each line the description of the process in italics, for the discussion later.

```
accton    e =    6, chars =      0, stat =   0:     S
sh        e = 2106, chars =  15632, stat =   0:     S
dd        e =    8, chars = 273344, stat =   0:          second child
a.out     e =  202, chars =    921, stat =   0:          parent
a.out     e =  407, chars =      0, stat = 134:     F    first child
a.out     e =  600, chars =      0, stat =   9:     F    fourth child
a.out     e =  801, chars =      0, stat =   0:     F    third child
```

The elapsed time values are measured in units of clock ticks per second. From Figure 2.14, the value on this system is 100. For example, the `sleep(2)` in the parent corresponds to the elapsed time of 202 clock ticks. For the first child, the `sleep(4)` becomes 407 clock ticks. Note that the amount of time a process sleeps is not exact. (We'll return to the `sleep` function in Chapter 10.) Also, the calls to `fork` and `exit` take some amount of time.

Note that the `ac_stat` member is not the true termination status of the process, but corresponds to a portion of the termination status that we discussed in Section 8.6. The only information in this byte is a core-flag bit (usually the high-order bit) and the signal number (usually the seven low-order bits), if the process terminated abnormally. If the process terminated normally, we are not able to obtain the exit status from the accounting file. For the first child, this value is 128 + 6. The 128 is the core flag bit, and 6 happens to be the value on this system for SIGABRT, which is generated by the call to abort. The value 9 for the fourth child corresponds to the value of SIGKILL. We can't tell from the accounting data that the parent's argument to exit was 2 and that the third child's argument to exit was 0.

The size of the file /etc/termcap that the dd process copies in the second child is 136,663 bytes. The number of characters of I/O is just over twice this value. It is twice the value, as 136,663 bytes are read in, then 136,663 bytes are written out. Even though the output goes to the null device, the bytes are still accounted for.

The ac_flag values are as we expect. The F flag is set for all the child processes except the second child, which does the execl. The F flag is not set for the parent, because the interactive shell that executed the parent did a fork and then an exec of the a.out file. The first child process calls abort, which generates a SIGABRT signal to generate the core dump. Note that neither the X flag nor the D flag is on, as they are not supported on Solaris; the information they represent can be derived from the ac_stat field. The fourth child also terminates because of a signal, but the SIGKILL signal does not generate a core dump; it only terminates the process.

As a final note, the first child has a 0 count for the number of characters of I/O, yet this process generated a core file. It appears that the I/O required to write the core file is not charged to the process. □

8.15 User Identification

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call getpwuid(getuid()), but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in under (Section 6.8), and the getlogin function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);

                    Returns: pointer to string giving login name if OK, NULL on error
```

This function can fail if the process is not attached to a terminal that a user logged in to. We normally call these processes *daemons*. We discuss them in Chapter 13.

Given the login name, we can then use it to look up the user in the password file—to determine the login shell, for example—using getpwnam.

To find the login name, UNIX systems have historically called the `ttyname` function (Section 18.9) and then tried to find a matching entry in the `utmp` file (Section 6.8). FreeBSD and Mac OS X store the login name in the session structure associated with the process table entry and provide system calls to fetch and store this name.

System V provided the `cuserid` function to return the login name. This function called `getlogin` and, if that failed, did a `getpwuid(getuid())`. The IEEE Standard 1003.1−1988 specified `cuserid`, but it called for the effective user ID to be used, instead of the real user ID. The 1990 version of POSIX.1 dropped the `cuserid` function.

The environment variable `LOGNAME` is usually initialized with the user's login name by `login(1)` and inherited by the login shell. Realize, however, that a user can modify an environment variable, so we shouldn't use `LOGNAME` to validate the user in any way. Instead, `getlogin` should be used.

8.16 Process Times

In Section 1.10, we described three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

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

clock_t times(struct tms *buf);

                    Returns: elapsed wall clock time in clock ticks if OK, −1 on error
```

This function fills in the `tms` structure pointed to by *buf*:

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value. For example, we call `times` and save the return value. At some later time, we call `times` again and subtract the earlier return value from the new return value. The difference is the wall clock time. (It is possible, though unlikely, for a long-running process to overflow the wall clock time; see Exercise 1.6.)

The two structure fields for child processes contain values only for children that we have waited for with `wait`, `waitid`, or `waitpid`.

All the `clock_t` values returned by this function are converted to seconds using the number of clock ticks per second—the `_SC_CLK_TCK` value returned by `sysconf` (Section 2.5.4).

Most implementations provide the getrusage(2) function. This function returns the CPU times and 14 other values indicating resource usage. Historically, this function originated with the BSD operating system, so BSD-derived implementations generally support more of the fields than do other implementations.

Example

The program in Figure 8.30 executes each command-line argument as a shell command string, timing the command and printing the values from the tms structure.

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

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

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

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)        /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1)   /* starting values */
        err_sys("times error");

    if ((status = system(cmd)) < 0)     /* execute command */
        err_sys("system() error");

    if ((end = times(&tmsend)) == -1)       /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
```

```
            err_sys("sysconf error");
    printf("  real:  %7.2f\n", real / (double) clkctck);
    printf("  user:  %7.2f\n",
      (tmsend->tms_utime - tmsstart->tms_utime) / (double) clkctck);
    printf("  sys:   %7.2f\n",
      (tmsend->tms_stime - tmsstart->tms_stime) / (double) clkctck);
    printf("  child user:  %7.2f\n",
      (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clkctck);
    printf("  child sys:   %7.2f\n",
      (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clkctck);
}
```

Figure 8.30 Time and execute all command-line arguments

If we run this program, we get

```
$ ./a.out "sleep 5" "date"

command: sleep 5
  real:     5.02
  user:     0.00
  sys:      0.00
  child user:     0.01
  child sys:      0.00
normal termination, exit status = 0

command: date
Mon Mar 22 00:43:58 EST 2004
  real:     0.01
  user:     0.00
  sys:      0.00
  child user:     0.01
  child sys:      0.00
normal termination, exit status = 0
```

In these two examples, all the CPU time appears in the child process, which is where the shell and the command execute. □

8.17 Summary

A thorough understanding of the UNIX System's process control is essential for advanced programming. There are only a few functions to master: `fork`, the `exec` family, `_exit`, `wait`, and `waitpid`. These primitives are used in many applications. The `fork` function also gave us an opportunity to look at race conditions.

Our examination of the `system` function and process accounting gave us another look at all these process control functions. We also looked at another variation of the `exec` functions: interpreter files and how they operate. An understanding of the various user IDs and group IDs that are provided—real, effective, and saved—is critical to writing safe set-user-ID programs.

Given an understanding of a single process and its children, in the next chapter we examine the relationship of a process to other processes—sessions and job control. We then complete our discussion of processes in Chapter 10 when we describe signals.

Exercises

8.1 In Figure 8.3, we said that replacing the call to `_exit` with a call to `exit` might cause the standard output to be closed and `printf` to return −1. Modify the program to check whether your implementation behaves this way. If it does not, how can you simulate this behavior?

8.2 Recall the typical arrangement of memory in Figure 7.6. Because the stack frames corresponding to each function call are usually stored in the stack, and because after a `vfork`, the child runs in the address space of the parent, what happens if the call to `vfork` is from a function other than `main` and the child does a return from this function after the `vfork`? Write a test program to verify this, and draw a picture of what's happening.

8.3 When we execute the program in Figure 8.13 one time, as in

```
$ ./a.out
```

the output is correct. But if we execute the program multiple times, one right after the other, as in

```
$ ./a.out ; ./a.out ; ./a.out
output from parent
ooutput from parent
ouotuptut from child
put from parent
output from child
utput from child
```

the output is not correct. What's happening? How can we correct this? Can this problem happen if we let the child write its output first?

8.4 In the program shown in Figure 8.20, we call `execl`, specifying the *pathname* of the interpreter file. If we called `execlp` instead, specifying a *filename* of `testinterp`, and if the directory `/home/sar/bin` was a path prefix, what would be printed as `argv[2]` when the program is run?

8.5 How can a process obtain its saved set-user-ID?

8.6 Write a program that creates a zombie, and then call `system` to execute the `ps(1)` command to verify that the process is a zombie.

8.7 We mentioned in Section 8.10 that POSIX.1 requires that open directory streams be closed across an `exec`. Verify this as follows: call `opendir` for the root directory, peek at your system's implementation of the `DIR` structure, and print the close-on-exec flag. Then open the same directory for reading, and print the close-on-exec flag.

9

Process Relationships

9.1 Introduction

We learned in the previous chapter that there are relationships between processes. First, every process has a parent process (the initial kernel-level process is usually its own parent). The parent is notified when the child terminates, and the parent can obtain the child's exit status. We also mentioned process groups when we described the `waitpid` function (Section 8.6) and how we can wait for any process in a process group to terminate.

In this chapter, we'll look at process groups in more detail and the concept of sessions that was introduced by POSIX.1. We'll also look at the relationship between the login shell that is invoked for us when we log in and all the processes that we start from our login shell.

It is impossible to describe these relationships without talking about signals, and to talk about signals, we need many of the concepts in this chapter. If you are unfamiliar with the UNIX System signal mechanism, you may want to skim through Chapter 10 at this point.

9.2 Terminal Logins

Let's start by looking at the programs that are executed when we log in to a UNIX system. In early UNIX systems, such as Version 7, users logged in using dumb terminals that were connected to the host with hard-wired connections. The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel. For example, the common devices on PDP-11s were DH-11s and DZ-11s. A host had a fixed number of these terminal devices, so there was a known upper limit on the number of simultaneous logins.

As bit-mapped graphical terminals became available, windowing systems were developed to provide users with new ways to interact with host computers. Applications were developed to create "terminal windows" to emulate character-based terminals, allowing users to interact with hosts in familiar ways (i.e., via the shell command line).

Today, some platforms allow you to start a windowing system after logging in, whereas other platforms automatically start the windowing system for you. In the latter case, you might still have to log in, depending on how the windowing system is configured (some windowing systems can be configured to log you in automatically).

The procedure that we now describe is used to log in to a UNIX system using a terminal. The procedure is similar regardless of the type of terminal we use—it could be a character-based terminal, a graphical terminal emulating a simple character-based terminal, or a graphical terminal running a windowing system.

BSD Terminal Logins

This procedure has not changed much over the past 30 years. The system administrator creates a file, usually `/etc/ttys`, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the `getty` program. One parameter is the baud rate of the terminal, for example. When the system is bootstrapped, the kernel creates process ID 1, the `init` process, and it is `init` that brings the system up multiuser. The `init` process reads the file `/etc/ttys` and, for every terminal device that allows a login, does a `fork` followed by an `exec` of the program `getty`. This gives us the processes shown in Figure 9.1.

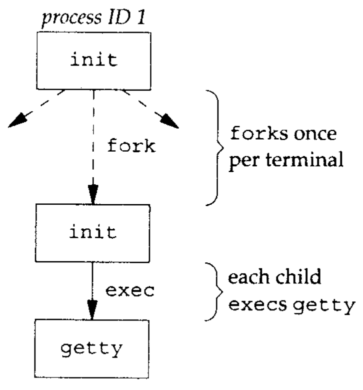

Figure 9.1 Processes invoked by `init` to allow terminal logins

All the processes shown in Figure 9.1 have a real user ID of 0 and an effective user ID of 0 (i.e., they all have superuser privileges). The `init` process also `exec`s the `getty` program with an empty environment.

It is getty that calls open for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the open may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then getty outputs something like login: and waits for us to enter our user name. If the terminal supports multiple speeds, getty can detect special characters that tell it to change the terminal's speed (baud rate). Consult your UNIX system manuals for additional details on the getty program and the data files (gettytab) that can drive its actions.

When we enter our user name, getty's job is complete, and it then invokes the login program, similar to

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

(There can be options in the gettytab file to have it invoke other programs, but the default is the login program.) init invokes getty with an empty environment; getty creates an environment for login (the envp argument) with the name of the terminal (something like TERM=foo, where the type of terminal foo is taken from the gettytab file) and any environment strings that are specified in the gettytab. The -p flag to login tells it to preserve the environment that it is passed and to add to that environment, not replace it. Figure 9.2 shows the state of these processes right after login has been invoked.

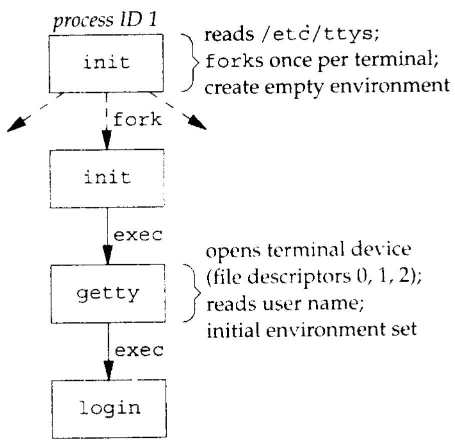

Figure 9.2 State of processes after login has been invoked

All the processes shown in Figure 9.2 have superuser privileges, since the original init process has superuser privileges. The process ID of the bottom three processes in Figure 9.2 is the same, since the process ID does not change across an exec. Also, all the processes other than the original init process have a parent process ID of 1.

The login program does many things. Since it has our user name, it can call getpwnam to fetch our password file entry. Then login calls getpass(3) to display the prompt Password: and read our password (with echoing disabled, of course). It calls crypt(3) to encrypt the password that we entered and compares the encrypted result to the pw_passwd field from our shadow password file entry. If the login attempt fails because of an invalid password (after a few tries), login calls exit with an argument of 1. This termination will be noticed by the parent (init), and it will do another fork followed by an exec of getty, starting the procedure over again for this terminal.

This is the traditional authentication procedure used on UNIX systems. Modern UNIX systems have evolved to support multiple authentication procedures. For example, FreeBSD, Linux, Mac OS X, and Solaris all support a more flexible scheme known as PAM (Pluggable Authentication Modules). PAM allows an administrator to configure the authentication methods to be used to access services that are written to use the PAM library.

If our application needs to verify that a user has the appropriate permission to perform a task, we can either hard code the authentication mechanism in the application, or we can use the PAM library to give us the equivalent functionality. The advantage to using PAM is that administrators can configure different ways to authenticate users for different tasks, based on the local site policies.

If we log in correctly, login will

- Change to our home directory (chdir)
- Change the ownership of our terminal device (chown) so we own it
- Change the access permissions for our terminal device so we have permission to read from and write to it
- Set our group IDs by calling setgid and initgroups
- Initialize the environment with all the information that login has: our home directory (HOME), shell (SHELL), user name (USER and LOGNAME), and a default path (PATH)
- Change to our user ID (setuid) and invoke our login shell, as in

```
execl("/bin/sh", "-sh", (char *)0);
```

The minus sign as the first character of argv[0] is a flag to all the shells that they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

The login program really does more than we've described here. It optionally prints the message-of-the-day file, checks for new mail, and performs other tasks. We're interested only in the features that we've described.

Recall from our discussion of the setuid function in Section 8.11 that since it is called by a superuser process, setuid changes all three user IDs: the real user ID, effective user ID, and saved set-user-ID. The call to setgid that was done earlier by login has the same effect on all three group IDs.

At this point, our login shell is running. Its parent process ID is the original init process (process ID 1), so when our login shell terminates, init is notified (it is sent a SIGCHLD signal), and it can start the whole procedure over again for this terminal. File descriptors 0, 1, and 2 for our login shell are set to the terminal device. Figure 9.3 shows this arrangement.

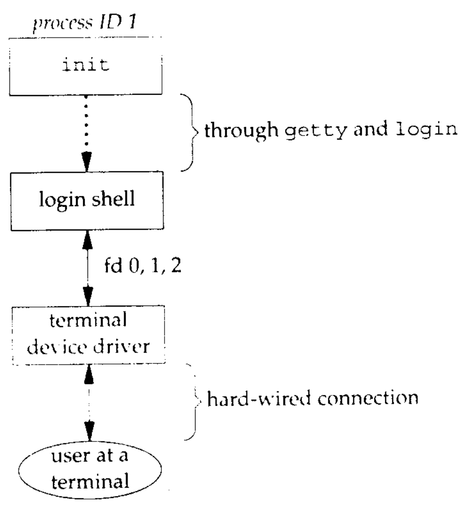

Figure 9.3 Arrangement of processes after everything is set for a terminal login

Our login shell now reads its start-up files (`.profile` for the Bourne shell and Korn shell; `.bash_profile`, `.bash_login`, or `.profile` for the GNU Bourne-again shell; and `.cshrc` and `.login` for the C shell). These start-up files usually change some of the environment variables and add many additional variables to the environment. For example, most users set their own PATH and often prompt for the actual terminal type (TERM). When the start-up files are done, we finally get the shell's prompt and can enter commands.

Mac OS X Terminal Logins

On Mac OS X, the terminal login process follows the same steps as in the BSD login process, since Mac OS X is based in part on FreeBSD. With Mac OS X, however, we are presented with a graphical-based login screen from the start.

Linux Terminal Logins

The Linux login procedure is very similar to the BSD procedure. Indeed, the Linux `login` command is derived from the 4.3BSD `login` command. The main difference between the BSD login procedure and the Linux login procedure is in the way the terminal configuration is specified.

On Linux, `/etc/inittab` contains the configuration information specifying the terminal devices for which `init` should start a `getty` process, similar to the way it is done on System V. Depending on the version of `getty` in use, the terminal characteristics are specified either on the command line (as with `agetty`) or in the file `/etc/gettydefs` (as with `mgetty`).

Solaris Terminal Logins

Solaris supports two forms of terminal logins: (a) `getty` style, as described previously for BSD, and (b) `ttymon` logins, a feature introduced with SVR4. Normally, `getty` is used for the console, and `ttymon` is used for other terminal logins.

The `ttymon` command is part of a larger facility termed SAF, the Service Access Facility. The goal of the SAF was to provide a consistent way to administer services that provide access to a system. (See Chapter 6 of Rago [1993] for more details.) For our purposes, we end up with the same picture as in Figure 9.3, with a different set of steps between `init` and the login shell. `init` is the parent of `sac` (the service access controller), which does a `fork` and `exec` of the `ttymon` program when the system enters multiuser state. The `ttymon` program monitors all the terminal ports listed in its configuration file and does a `fork` when we've entered our login name. This child of `ttymon` does an `exec` of `login`, and `login` prompts us for our password. Once this is done, `login` `exec`s our login shell, and we're at the position shown in Figure 9.3. One difference is that the parent of our login shell is now `ttymon`, whereas the parent of the login shell from a `getty` login is `init`.

9.3 Network Logins

The main (physical) difference between logging in to a system through a serial terminal and logging in to a system through a network is that the connection between the terminal and the computer isn't point-to-point. In this case, `login` is simply a service available, just like any other network service, such as FTP or SMTP.

With the terminal logins that we described in the previous section, `init` knows which terminal devices are enabled for logins and spawns a `getty` process for each device. In the case of network logins, however, all the logins come through the kernel's network interface drivers (e.g., the Ethernet driver), and we don't know ahead of time how many of these will occur. Instead of having a process waiting for each possible login, we now have to wait for a network connection request to arrive.

To allow the same software to process logins over both terminal logins and network logins, a software driver called a *pseudo terminal* is used to emulate the behavior of a serial terminal and map terminal operations to network operations, and vice versa. (In Chapter 19, we'll talk about pseudo terminals in detail.)

BSD Network Logins

In BSD, a single process waits for most network connections: the `inetd` process, sometimes called the *Internet superserver*. In this section, we'll look at the sequence of processes involved in network logins for a BSD system. We are not interested in the detailed network programming aspects of these processes; refer to Stevens, Fenner, and Rudoff [2004] for all the details.

As part of the system start-up, `init` invokes a shell that executes the shell script `/etc/rc`. One of the daemons that is started by this shell script is `inetd`. Once the shell script terminates, the parent process of `inetd` becomes `init`; `inetd` waits for

TCP/IP connection requests to arrive at the host. When a connection request arrives for it to handle, `inetd` does a `fork` and `exec` of the appropriate program.

Let's assume that a TCP connection request arrives for the TELNET server. TELNET is a remote login application that uses the TCP protocol. A user on another host (that is connected to the server's host through a network of some form) or on the same host initiates the login by starting the TELNET client:

`telnet` *hostname*

The client opens a TCP connection to *hostname*, and the program that's started on *hostname* is called the TELNET server. The client and the server then exchange data across the TCP connection using the TELNET application protocol. What has happened is that the user who started the client program is now logged in to the server's host. (This assumes, of course, that the user has a valid account on the server's host.) Figure 9.4 shows the sequence of processes involved in executing the TELNET server, called `telnetd`.

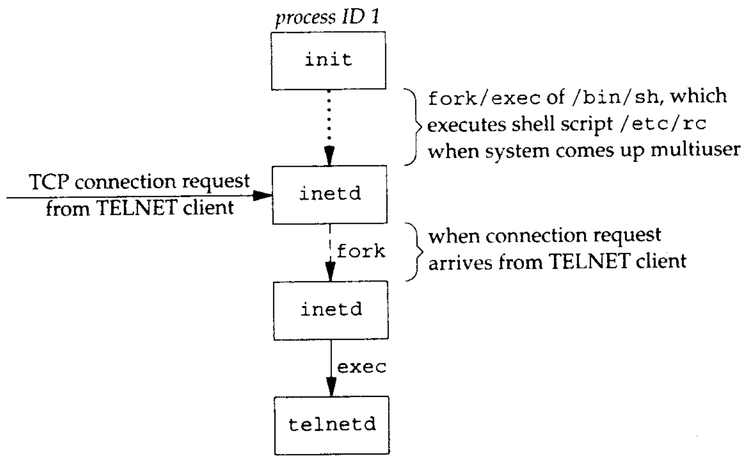
process ID 1
init
fork/exec of /bin/sh, which executes shell script /etc/rc when system comes up multiuser
TCP connection request from TELNET client
inetd
fork
when connection request arrives from TELNET client
inetd
exec
telnetd

Figure 9.4 Sequence of processes involved in executing TELNET server

The `telnetd` process then opens a pseudo-terminal device and splits into two processes using `fork`. The parent handles the communication across the network connection, and the child does an `exec` of the `login` program. The parent and the child are connected through the pseudo terminal. Before doing the `exec`, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, `login` performs the same steps we described in Section 9.2: it changes to our home directory and sets our group IDs, user ID, and our initial environment. Then `login` replaces itself with our login shell by calling `exec`. Figure 9.5 shows the arrangement of the processes at this point.

Obviously, a lot is going on between the pseudo-terminal device driver and the actual user at the terminal. We'll show all the processes involved in this type of arrangement in Chapter 19 when we talk about pseudo terminals in more detail.

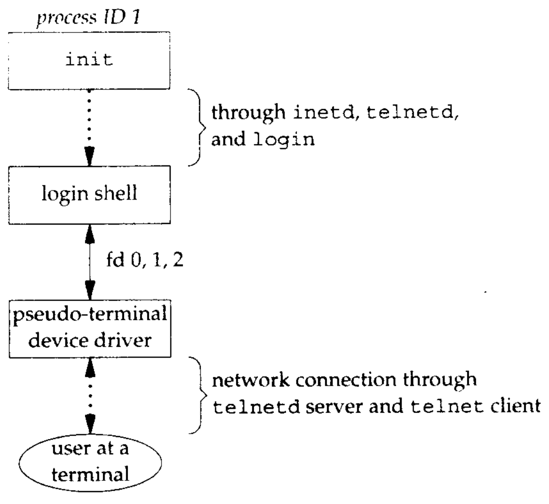
process ID 1
init
through inetd, telnetd, and login
login shell
fd 0, 1, 2
pseudo-terminal device driver
network connection through telnetd server and telnet client
user at a terminal

Figure 9.5 Arrangement of processes after everything is set for a network login

The important thing to understand is that whether we log in through a terminal (Figure 9.3) or a network (Figure 9.5), we have a login shell with its standard input, standard output, and standard error connected to either a terminal device or a pseudo-terminal device. We'll see in the coming sections that this login shell is the start of a POSIX.1 session, and that the terminal or pseudo terminal is the controlling terminal for the session.

Mac OS X Network Logins

Logging in to a Mac OS X system over a network is identical to a BSD system, because Mac OS X is based partially on FreeBSD.

Linux Network Logins

Network logins under Linux are the same as under BSD, except that an alternate `inetd` process is used, called the extended Internet services daemon, `xinetd`. The `xinetd` process provides a finer level of control over services it starts than does `inetd`.

Solaris Network Logins

The scenario for network logins under Solaris is almost identical to the steps under BSD and Linux. An `inetd` server is used similar to the BSD version. The Solaris version has the additional ability to run under the service access facility framework, although it is not configured to do so. Instead, the `inetd` server is started by `init`. Either way, we end up with the same overall picture as in Figure 9.5.

9.4 Process Groups

In addition to having a process ID, each process also belongs to a process group. We'll encounter process groups again when we discuss signals in Chapter 10.

A process group is a collection of one or more processes, usually associated with the same job (job control is discussed in Section 9.8), that can receive signals from the same terminal. Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a `pid_t` data type. The function `getpgrp` returns the process group ID of the calling process.

```
#include <unistd.h>

pid_t getpgrp(void);

                    Returns: process group ID of calling process
```

In older BSD-derived systems, the `getpgrp` function took a *pid* argument and returned the process group for that process. The Single UNIX Specification defines the `getpgid` function as an XSI extension that mimics this behavior.

```
#include <unistd.h>

pid_t getpgid(pid_t pid);

                    Returns: process group ID if OK, -1 on error
```

If *pid* is 0, the process group ID of the calling process is returned. Thus,

```
getpgid(0);
```

is equivalent to

```
getpgrp();
```

Each process group can have a process group leader. The leader is identified by its process group ID being equal to its process ID.

It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates. This is called the process group lifetime—the period of time that begins when the group is created and ends when the last remaining process leaves the group. The last remaining process in the process group can either terminate or enter some other process group.

A process joins an existing process group or creates a new process group by calling `setpgid`. (In the next section, we'll see that `setsid` also creates a new process group.)

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);

                                Returns: 0 if OK, -1 on error
```

This function sets the process group ID to *pgid* in the process whose process ID equals *pid*. If the two arguments are equal, the process specified by *pid* becomes a process group leader. If *pid* is 0, the process ID of the caller is used. Also, if *pgid* is 0, the process ID specified by *pid* is used as the process group ID.

A process can set the process group ID of only itself or any of its children. Furthermore, it can't change the process group ID of one of its children after that child has called one of the exec functions.

In most job-control shells, this function is called after a fork to have the parent set the process group ID of the child, and to have the child set its own process group ID. One of these calls is redundant, but by doing both, we are guaranteed that the child is placed into its own process group before either process assumes that this has happened. If we didn't do this, we would have a race condition, since the child's process group membership would depend on which process executes first.

When we discuss signals, we'll see how we can send a signal to either a single process (identified by its process ID) or a process group (identified by its process group ID). Similarly, the waitpid function from Section 8.6 lets us wait for either a single process or one process from a specified process group.

9.5 Sessions

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure 9.6. Here we have three process groups in a single session.

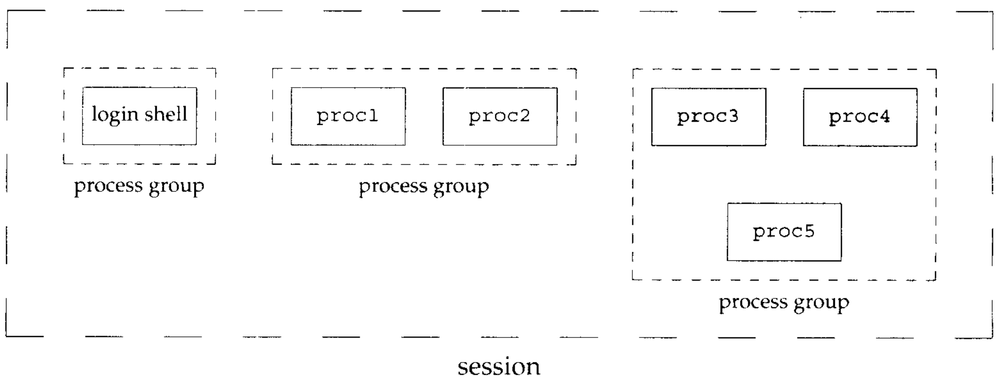

Figure 9.6 Arrangement of processes into process groups and sessions

The processes in a process group are usually placed there by a shell pipeline. For example, the arrangement shown in Figure 9.6 could have been generated by shell commands of the form

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

A process establishes a new session by calling the `setsid` function.

```
#include <unistd.h>

pid_t setsid(void);

                              Returns: process group ID if OK, -1 on error
```

If the calling process is not a process group leader, this function creates a new session. Three things happen.

1. The process becomes the *session leader* of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
2. The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
3. The process has no controlling terminal. (We'll discuss controlling terminals in the next section.) If the process had a controlling terminal before calling `setsid`, that association is broken.

This function returns an error if the caller is already a process group leader. To ensure this is not the case, the usual practice is to call `fork` and have the parent terminate and the child continue. We are guaranteed that the child is not a process group leader, because the process group ID of the parent is inherited by the child, but the child gets a new process ID. Hence, it is impossible for the child's process ID to equal its inherited process group ID.

The Single UNIX Specification talks only about a "session leader." There is no "session ID" similar to a process ID or a process group ID. Obviously, a session leader is a single process that has a unique process ID, so we could talk about a session ID that is the process ID of the session leader. This concept of a session ID was introduced in SVR4. Historically, BSD-based systems didn't support this notion, but have since been updated to include it. The `getsid` function returns the process group ID of a process's session leader. The `getsid` function is included as an XSI extension in the Single UNIX Specification.

Some implementations, such as Solaris, join with the Single UNIX Specification in the practice of avoiding the use of the phrase "session ID," opting instead to refer to this as the "process group ID of the session leader." The two are equivalent, since the session leader is always the leader of a process group.

```
#include <unistd.h>

pid_t getsid(pid_t pid);

              Returns: session leader's process group ID if OK, -1 on error
```

If *pid* is 0, `getsid` returns the process group ID of the calling process's session leader. For security reasons, some implementations may restrict the calling process from obtaining the process group ID of the session leader if *pid* doesn't belong to the same session as the caller.

9.6 Controlling Terminal

Sessions and process groups have a few other characteristics.

- A session can have a single *controlling terminal*. This is usually the terminal device (in the case of a terminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the *controlling process*.
- The process groups within a session can be divided into a single *foreground process group* and one or more *background process groups*.
- If a session has a controlling terminal, it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.
- Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

These characteristics are shown in Figure 9.7.

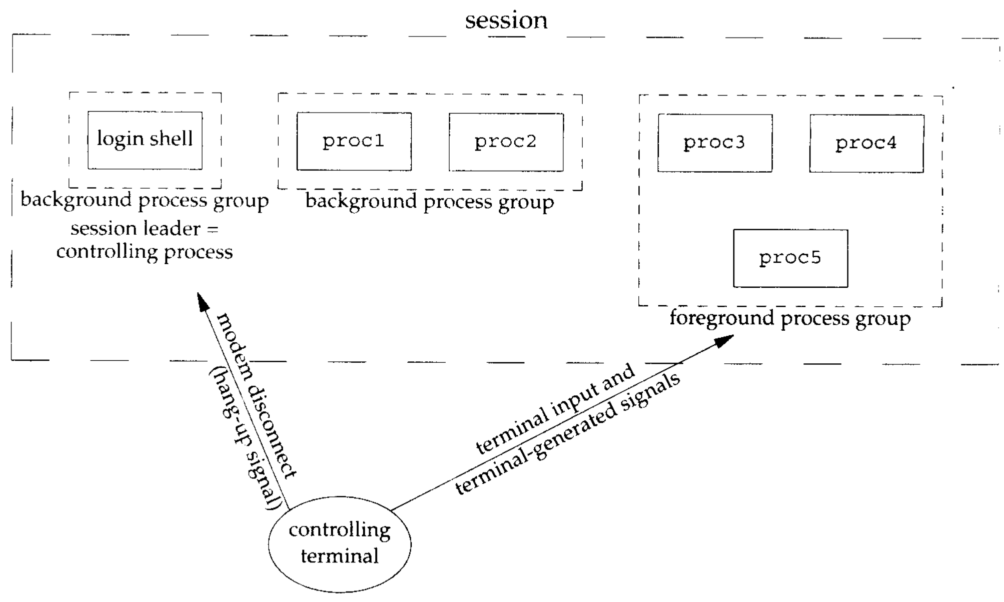

Figure 9.7 Process groups and sessions showing controlling terminal

Usually, we don't have to worry about the controlling terminal; it is established automatically when we log in.

POSIX.1 leaves the choice of the mechanism used to allocate a controlling terminal up to each individual implementation. We'll show the actual steps in Section 19.4.

Systems derived from UNIX System V allocate the controlling terminal for a session when the session leader opens the first terminal device that is not already associated with a session. This assumes that the call to `open` by the session leader does not specify the `O_NOCTTY` flag (Section 3.3).

BSD-based systems allocate the controlling terminal for a session when the session leader calls `ioctl` with a *request* argument of `TIOCSCTTY` (the third argument is a null pointer). The session cannot already have a controlling terminal for this call to succeed. (Normally, this call to `ioctl` follows a call to `setsid`, which guarantees that the process is a session leader without a controlling terminal.) The POSIX.1 `O_NOCTTY` flag to `open` is not used by BSD-based systems, except in compatibility-mode support for other systems.

There are times when a program wants to talk to the controlling terminal, regardless of whether the standard input or standard output is redirected. The way a program guarantees that it is talking to the controlling terminal is to `open` the file `/dev/tty`. This special file is a synonym within the kernel for the controlling terminal. Naturally, if the program doesn't have a controlling terminal, the `open` of this device will fail.

The classic example is the `getpass`(3) function, which reads a password (with terminal echoing turned off, of course). This function is called by the `crypt`(1) program and can be used in a pipeline. For example,

```
crypt < salaries | lpr
```

decrypts the file `salaries` and pipes the output to the print spooler. Because `crypt` reads its input file on its standard input, the standard input can't be used to enter the password. Also, `crypt` is designed so that we have to enter the encryption password each time we run the program, to prevent us from saving the password in a file (which could be a security hole).

There are known ways to break the encoding used by the `crypt` program. See Garfinkel et al. [2003] for more details on encrypting files.

9.7 `tcgetpgrp`, `tcsetpgrp`, and `tcgetsid` Functions

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals (Figure 9.7).

```
#include <unistd.h>

pid_t tcgetpgrp(int filedes);

        Returns: process group ID of foreground process group if OK, −1 on error

int tcsetpgrp(int filedes, pid_t pgrpid);

                                              Returns: 0 if OK, −1 on error
```

The function `tcgetpgrp` returns the process group ID of the foreground process group associated with the terminal open on *filedes*.

If the process has a controlling terminal, the process can call `tcsetpgrp` to set the foreground process group ID to *pgrpid*. The value of *pgrpid* must be the process group ID of a process group in the same session, and *filedes* must refer to the controlling terminal of the session.

Most applications don't call these two functions directly. They are normally called by job-control shells.

The Single UNIX Specification defines an XSI extension called `tcgetsid` to allow an application to obtain the process group ID for the session leader given a file descriptor for the controlling TTY.

```
#include <termios.h>

pid_t tcgetsid(int filedes);
```

Returns: session leader's process group ID if OK, –1 on error

Applications that need to manage controlling terminals can use `tcgetsid` to identify the session ID of the controlling terminal's session leader (which is equivalent to the session-leader's process group ID).

9.8 Job Control

Job control is a feature added to BSD around 1980. This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support:

1. A shell that supports job control
2. The terminal driver in the kernel must support job control
3. The kernel must support certain job-control signals

SVR3 provided a different form of job control called *shell layers*. The BSD form of job control, however, was selected by POSIX.1 and is what we describe here. In earlier versions of the standard, job control support was optional, but POSIX.1 now requires platforms to support it.

From our perspective, using job control from a shell, we can start a job in either the foreground or the background. A job is simply a collection of processes, often a pipeline of processes. For example,

```
vi main.c
```

starts a job consisting of one process in the foreground. The commands

```
pr *.c | lpr &
make all &
```

start two jobs in the background. All the processes invoked by these background jobs are in the background.

As we said, to use the features provided by job control, we need to be using a shell that supports job control. With older systems, it was simple to say which shells supported job control and which didn't. The C shell supported job control, the Bourne shell didn't, and it was an option with the Korn shell, depending whether the host supported job control. But the C shell has been ported to systems (e.g., earlier versions of System V) that don't support job control, and the SVR4 Bourne shell, when invoked by the name `jsh` instead of `sh`, supports job control. The Korn shell continues to support job control if the host does. The Bourne-again shell also supports job control. We'll just talk generically about a shell that supports job control, versus one that doesn't, when the difference between the various shells doesn't matter.

When we start a background job, the shell assigns it a job identifier and prints one or more of the process IDs. The following script shows how the Korn shell handles this:

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                               just press RETURN
[2] +  Done             pr *.c | lpr &
[1] +  Done             make all > Make.out &
```

The `make` is job number 1 and the starting process ID is 1475. The next pipeline is job number 2 and the process ID of the first process is 1490. When the jobs are done and when we press RETURN, the shell tells us that the jobs are complete. The reason we have to press RETURN is to have the shell print its prompt. The shell doesn't print the changed status of background jobs at any random time—only right before it prints its prompt, to let us enter a new command line. If the shell didn't do this, it could output while we were entering an input line.

The interaction with the terminal driver arises because a special terminal character affects the foreground job: the suspend key (typically Control-Z). Entering this character causes the terminal driver to send the `SIGTSTP` signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver looks for three special characters, which generate signals to the foreground process group.

- The interrupt character (typically DELETE or Control-C) generates `SIGINT`.
- The quit character (typically Control-backslash) generates `SIGQUIT`.
- The suspend character (typically Control-Z) generates `SIGTSTP`.

In Chapter 18, we'll see how we can change these three characters to be any characters we choose and how we can disable the terminal driver's processing of these special characters.

Another job control condition can arise that must be handled by the terminal driver. Since we can have a foreground job and one or more background jobs, which of these receives the characters that we enter at the terminal? Only the foreground job receives terminal input. It is not an error for a background job to try to read from the terminal, but the terminal driver detects this and sends a special signal to the background job: `SIGTTIN`. This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal. The following demonstrates this:

```
$ cat > temp.foo &              start in background, but it'll read from standard input
[1]     1681
$                               we press RETURN
[1] + Stopped (SIGTTIN)           cat > temp.foo &
$ fg %1                         bring job number 1 into the foreground
cat > temp.foo                  the shell tells us which job is now in the foreground
hello, world                    enter one line
^D                              type the end-of-file character
$ cat temp.foo                  check that the one line was put into the file
hello, world
```

The shell starts the cat process in the background, but when cat tries to read its standard input (the controlling terminal), the terminal driver, knowing that it is a background job, sends the SIGTTIN signal to the background job. The shell detects this change in status of its child (recall our discussion of the wait and waitpid function in Section 8.6) and tells us that the job has been stopped. We then move the stopped job into the foreground with the shell's fg command. (Refer to the manual page for the shell that you are using, for all the details on its job control commands, such as fg and bg, and the various ways to identify the different jobs.) Doing this causes the shell to place the job into the foreground process group (tcsetpgrp) and send the continue signal (SIGCONT) to the process group. Since it is now in the foreground process group, the job can read from the controlling terminal.

What happens if a background job outputs to the controlling terminal? This is an option that we can allow or disallow. Normally, we use the stty(1) command to change this option. (We'll see in Chapter 18 how we can change this option from a program.) The following shows how this works:

```
$ cat temp.foo &           execute in background
[1]     1719
$ hello, world             the output from the background job appears after the prompt
                           we press RETURN
[1] +  Done         cat temp.foo &
$ stty tostop              disable ability of background jobs to output to controlling terminal
$ cat temp.foo &           try it again in the background
[1]     1721
$                          we press RETURN and find the job is stopped
[1] + Stopped(SIGTTOU)          cat temp.foo &
$ fg %1                    resume stopped job in the foreground
cat temp.foo               the shell tells us which job is now in the foreground
hello, world               and here is its output
```

When we disallow background jobs from writing to the controlling terminal, cat will block when it tries to write to its standard output, because the terminal driver identifies the write as coming from a background process and sends the job the SIGTTOU signal. As with the previous example, when we use the shell's fg command to bring the job into the foreground, the job completes.

Figure 9.8 summarizes some of the features of job control that we've been describing. The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process