7

Process Environment

7.1 Introduction

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

7.2 `main` Function

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

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

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

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

7.3 Process Termination

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

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

Abnormal termination occurs in three ways:

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

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

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

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

Exit Functions

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

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

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

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

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

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

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

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

```
exit(0);
```

is the same as

```
return(0);
```

from the `main` function.

Example

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

```
#include    <stdio.h>

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

Figure 7.1 Classic C program

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

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

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

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

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

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

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

□

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

`atexit` Function

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

```
#include <stdlib.h>

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

Returns: 0 if OK, nonzero on error

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

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

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

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

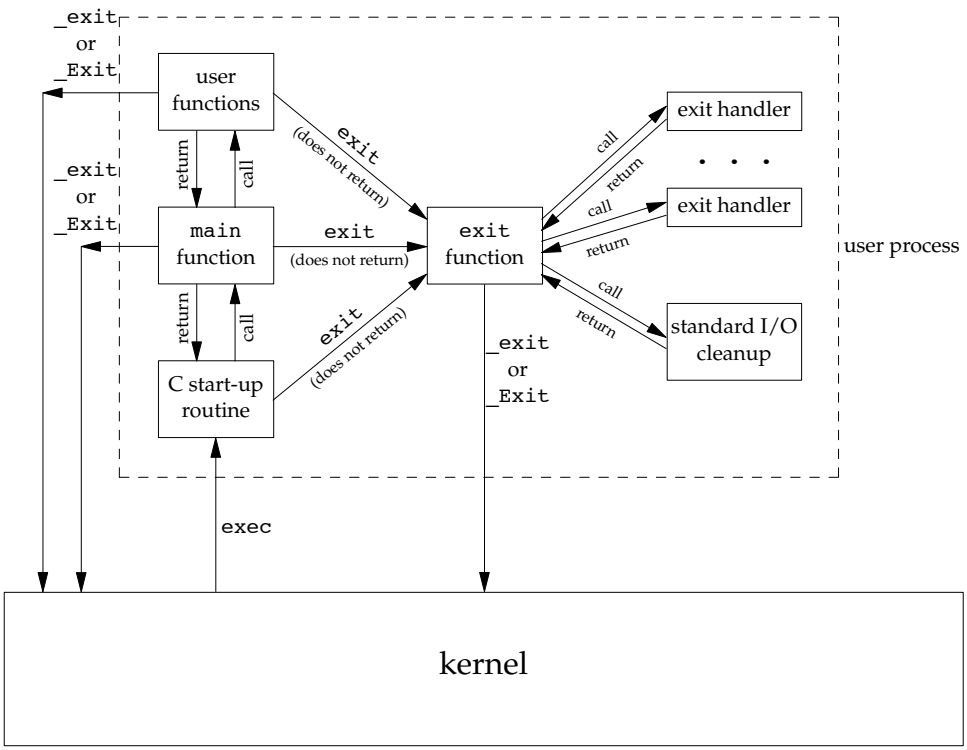

Figure 7.2 How a C program is started and how it terminates

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

Example

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

```
#include "apue.h"

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

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

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

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

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

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

Figure 7.3 Example of exit handlers

Executing the program in Figure 7.3 yields

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

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

7.4 Command-Line Arguments

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

Example

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

```
#include "apue.h"

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

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

Figure 7.4 Echo all command-line arguments to standard output

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

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

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

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

□

7.5 Environment List

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

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

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

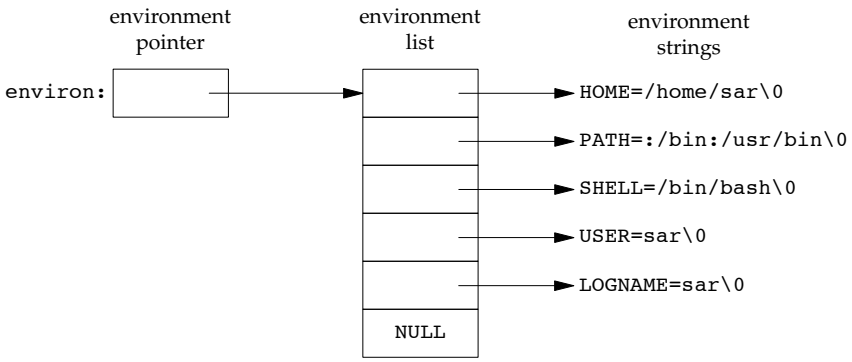

Figure 7.5 Environment consisting of five C character strings

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

By convention, the environment consists of

name=value

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

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

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

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

7.6 Memory Layout of a C Program

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

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

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

```
int   maxcount = 99;
```

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

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

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

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

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

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

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

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

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

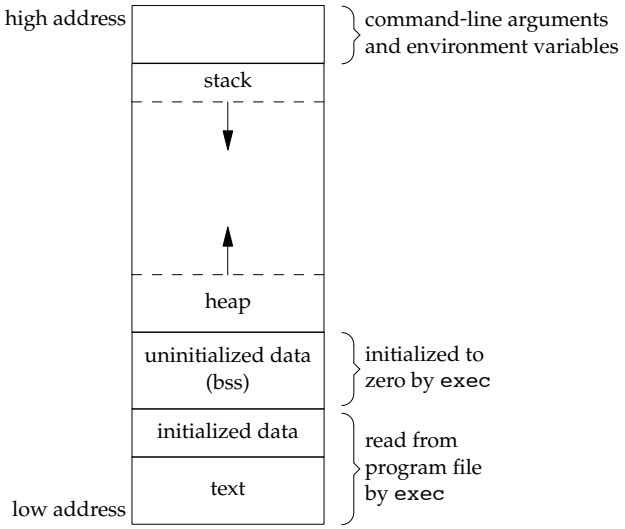

Figure 7.6 Typical memory arrangement

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

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

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

7.7 Shared Libraries

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

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

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

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

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

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

7.8 Memory Allocation

ISO C specifies three functions for memory allocation:

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

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

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

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

void free(void *ptr);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Alternate Memory Allocators

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

`libmalloc`

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

`vmalloc`

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

`quick-fit`

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

jemalloc

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

TCMalloc

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

alloca Function

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

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

7.9 Environment Variables

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

name=*value*

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

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

```
#include <stdlib.h>

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

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

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

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

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

Figure 7.7 Environment variables defined in the Single UNIX Specification

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

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

Figure 7.8 Support for various environment list functions

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

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

```
#include <stdlib.h>

int putenv(char *str);
```

Returns: 0 if OK, nonzero on error

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

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

Both return: 0 if OK, –1 on error

The operation of these three functions is as follows:

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

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

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

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

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

7.10 `setjmp` and `longjmp` Functions

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

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

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

```
#include "apue.h"

#define TOK_ADD    5

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

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

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

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

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

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

void
cmd_add(void)
{
    int     token;

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

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

Figure 7.9 Typical program skeleton for command processing

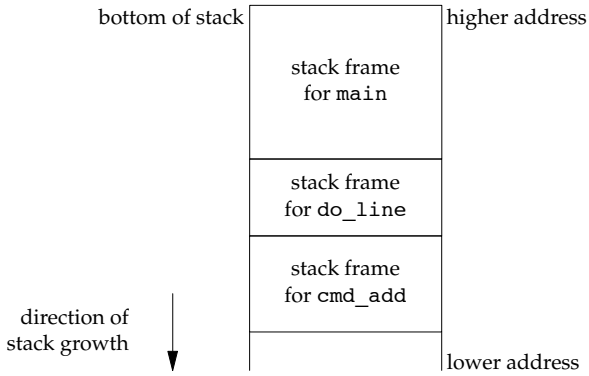

Figure 7.10 Stack frames after `cmd_add` has been called

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

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

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

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

```
#include <setjmp.h>

int setjmp(jmp_buf env);

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

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

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

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

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

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

#define TOK_ADD    5

jmp_buf jmpbuffer;

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

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

 . . .

void
cmd_add(void)
{
    int     token;

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

Figure 7.11 Example of setjmp and longjmp

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

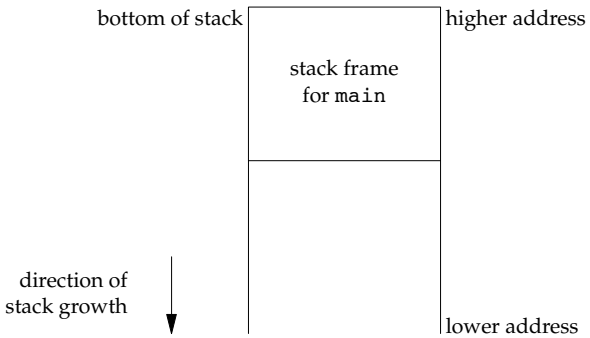

Figure 7.12 Stack frame after longjmp has been called

Automatic, Register, and Volatile Variables

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

Example

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

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

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

static jmp_buf  jmpbuffer;
static int      globval;

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

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

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

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

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

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

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

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

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

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

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

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

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

is equivalent to

```
"string1string2"
```

□

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

Potential Problem with Automatic Variables

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

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

```
#include    <stdio.h>

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

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

Figure 7.14 Incorrect usage of an automatic variable

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

7.11 `getrlimit` and `setrlimit` Functions

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

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

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

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

Both return: 0 if OK, –1 on error

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

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

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

Three rules govern the changing of the resource limits.

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

An infinite limit is specified by the constant RLIM_INFINITY.

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

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

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

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

Figure 7.15 Support for resource limits

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

Example

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

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

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

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

static void pr_limits(char *, int);

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

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

#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif

#ifdef RLIMIT_MSGQUEUE
    doit(RLIMIT_MSGQUEUE);
#endif

#ifdef RLIMIT_NICE
    doit(RLIMIT_NICE);
#endif

    doit(RLIMIT_NOFILE);

#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif

#ifdef RLIMIT_NPTS
    doit(RLIMIT_NPTS);
#endif

#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif

#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
```

```
#endif

#ifdef RLIMIT_SIGPENDING
    doit(RLIMIT_SIGPENDING);
#endif

    doit(RLIMIT_STACK);

#ifdef RLIMIT_SWAP
    doit(RLIMIT_SWAP);
#endif

#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif

    exit(0);
}

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

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

Figure 7.16 Print the current resource limits

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

```
doit(RLIMIT_CORE);
```

the C preprocessor expands this into

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

Running this program under FreeBSD gives us the following output:

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

Solaris gives us the following results:

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

□

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

7.12 Summary

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

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

Exercises

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

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

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

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

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

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

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

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

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

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

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

    if (val == 0) {
        int     val;

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