25

PROCESS TERMINATION

This chapter describes what happens when a process terminates. We begin by describing the use of *exit()* and *_exit()* to terminate a process. We then discuss the use of exit handlers to automatically perform cleanups when a process calls *exit()*. We conclude by considering some interactions between *fork()*, *stdio* buffers, and *exit()*.

25.1 Terminating a Process: *_exit()* and *exit()*

A process may terminate in two general ways. One of these is *abnormal* termination, caused by the delivery of a signal whose default action is to terminate the process (with or without a core dump), as described in Section 20.1. Alternatively, a process can terminate *normally*, using the *_exit()* system call.

```
#include <unistd.h>

void _exit(int status);
```

The *status* argument given to *_exit()* defines the *termination status* of the process, which is available to the parent of this process when it calls *wait()*. Although defined as an *int*, only the bottom 8 bits of *status* are actually made available to the parent. By convention, a termination status of 0 indicates that a process completed successfully, and a nonzero status value indicates that the process terminated unsuccessfully. There are no fixed rules about how nonzero status values are to be interpreted; different applications follow their own conventions, which should be described in their documentation. SUSv3 specifies two constants, `EXIT_SUCCESS` (0) and `EXIT_FAILURE` (1), that are used in most programs in this book.

A process is always successfully terminated by *_exit()* (i.e., *_exit()* never returns).

> Although any value in the range 0 to 255 can be passed to the parent via the *status* argument to *_exit()*, specifying values greater than 128 can cause confusion in shell scripts. The reason is that, when a command is terminated by a signal, the shell indicates this fact by setting the value of the variable *$?* to 128 plus the signal number, and this value is indistinguishable from that yielded when a process calls *_exit()* with the same *status* value.

Programs generally don't call *_exit()* directly, but instead call the *exit()* library function, which performs various actions before calling *_exit()*.

```
#include <stdlib.h>

void exit(int status);
```

The following actions are performed by *exit()*:

- Exit handlers (functions registered with *atexit()* and *on_exit()*) are called, in reverse order of their registration (Section 25.3).
- The *stdio* stream buffers are flushed.
- The *_exit()* system call is invoked, using the value supplied in *status*.

> Unlike *_exit()*, which is UNIX-specific, *exit()* is defined as part of the standard C library; that is, it is available with every C implementation.

One other way in which a process may terminate is to return from *main()*, either explicitly, or implicitly, by falling off the end of the *main()* function. Performing an explicit *return n* is generally equivalent to calling *exit(n)*, since the run-time function that invokes *main()* uses the return value from *main()* in a call to *exit()*.

> There is one circumstance in which calling *exit()* and returning from *main()* are not equivalent. If any steps performed during exit processing access variables local to *main()*, then doing a `return` from *main()* results in undefined behavior. For example, this could occur if a variable that is local to *main()* is specified in a call to *setvbuf()* or *setbuf()* (Section 13.2).

Performing a return without specifying a value, or falling off the end of the *main()* function, also results in the caller of *main()* invoking *exit()*, but with results that vary depending on the version of the C standard supported and the compilation options employed:

- In C89, the behavior in these circumstances is undefined; the program can terminate with an arbitrary *status* value. This is the behavior that occurs by default with *gcc* on Linux, where the exit status of the program is taken from some random value lying on the stack or in a particular CPU register. Terminating a program in this way should be avoided.

- The C99 standard requires that falling off the end of the main program should be equivalent to calling *exit(0)*. This is the behavior we obtain on Linux if we compile a program using *gcc –std=c99*.

25.2 Details of Process Termination

During both normal and abnormal termination of a process, the following actions occur:

- Open file descriptors, directory streams (Section 18.8), message catalog descriptors (see the *catopen(3)* and *catgets(3)* manual pages), and conversion descriptors (see the *iconv_open(3)* manual page) are closed.
- As a consequence of closing file descriptors, any file locks (Chapter 55) held by this process are released.
- Any attached System V shared memory segments are detached, and the *shm_nattch* counter corresponding to each segment is decremented by one. (Refer to Section 48.8.)
- For each System V semaphore for which a *semadj* value has been set by the process, that *semadj* value is added to the semaphore value. (Refer to Section 47.8.)
- If this is the controlling process for a controlling terminal, then the `SIGHUP` signal is sent to each process in the controlling terminal's foreground process group, and the terminal is disassociated from the session. We consider this point further in Section 34.6.
- Any POSIX named semaphores that are open in the calling process are closed as though *sem_close()* were called.
- Any POSIX message queues that are open in the calling process are closed as though *mq_close()* were called.
- If, as a consequence of this process exiting, a process group becomes orphaned and there are any stopped processes in that group, then all processes in the group are sent a `SIGHUP` signal followed by a `SIGCONT` signal. We consider this point further in Section 34.7.4.
- Any memory locks established by this process using *mlock()* or *mlockall()* (Section 50.2) are removed.
- Any memory mappings established by this process using *mmap()* are unmapped.

25.3 Exit Handlers

Sometimes, an application needs to automatically perform some operations on process termination. Consider the example of an application library that, if used during the life of the process, needs to have some cleanup actions performed automatically when the process exits. Since the library doesn't have control of when and how the process exits, and can't mandate that the main program call a library-specific cleanup function before exiting, cleanup is not guaranteed to occur. One approach in such situations is to use an *exit handler* (older System V manuals used the term *program termination routine*).

An exit handler is a programmer-supplied function that is registered at some point during the life of the process and is then automatically called during *normal* process termination via *exit()*. Exit handlers are not called if a program calls *_exit()* directly or if the process is terminated abnormally by a signal.

> To some extent, the fact that exit handlers are not called when a process is terminated by a signal limits their utility. The best we can do is to establish handlers for the signals that might be sent to the process, and have these handlers set a flag that causes the main program to call *exit()*. (Because *exit()* is not one of the async-signal-safe functions listed in Table 21-1, on page 426, we generally can't call it from a signal handler.) Even then, this doesn't handle the case of `SIGKILL`, whose default action can't be changed. This is one more reason we should avoid using `SIGKILL` to terminate a process (as noted in Section 20.2), and instead use `SIGTERM`, which is the default signal sent by the *kill* command.

Registering exit handlers

The GNU C library provides two ways of registering exit handlers. The first method, specified in SUSv3, is to use the *atexit()* function.

```
#include <stdlib.h>

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

Returns 0 on success, or nonzero on error

The *atexit()* function adds *func* to a list of functions that are called when the process terminates. The function *func* should be defined to take no arguments and return no value, thus having the following general form:

```
void
func(void)
{
    /* Perform some actions */
}
```

Note that *atexit()* returns a nonzero value (not necessarily –1) on error.

It is possible to register multiple exit handlers (and even the same exit handler multiple times). When the program invokes *exit()*, these functions are called *in reverse order* of registration. This ordering is logical because, typically, functions that are registered earlier are those that carry out more fundamental types of cleanups that may need to be performed after later-registered functions.

Essentially, any desired action can be performed inside an exit handler, including registering additional exit handlers, which are placed at the head of the list of exit handlers that remain to be called. However, if one of the exit handlers fails to return—either because it called *_exit()* or because the process was terminated by a signal (e.g., the exit handler called *raise()*)—then the remaining exit handlers are not called. In addition, the remaining actions that would normally be performed by *exit()* (i.e., flushing *stdio* buffers) are not performed.

> SUSv3 states that if an exit handler itself calls *exit()*, the results are undefined. On Linux, the remaining exit handlers are invoked as normal. However, on

some systems, this causes all of the exit handlers to once more be invoked, which can result in an infinite recursion (until a stack overflow kills the process). Portable applications should avoid calling *exit()* inside an exit handler.

SUSv3 requires that an implementation allow a process to be able to register at least 32 exit handlers. Using the call *sysconf(_SC_ATEXIT_MAX)*, a program can determine the implementation-defined upper limit on the number of exit handlers that can be registered. (However, there is no way to find out how many exit handlers have already been registered.) By chaining the registered exit handlers in a dynamically allocated linked list, *glibc* allows a virtually unlimited number of exit handlers to be registered. On Linux, *sysconf(_SC_ATEXIT_MAX)* returns 2,147,482,647 (i.e., the maximum signed 32-bit integer). In other words, something else will break (e.g., lack of memory) before we reach the limit on the number of functions that can be registered.

A child process created via *fork()* inherits a copy of its parent's exit handler registrations. When a process performs an *exec()*, all exit handler registrations are removed. (This is necessarily so, since an *exec()* replaces the code of the exit handlers along with the rest of the existing program code.)

We can't deregister an exit handler that has been registered with *atexit()* (or *on_exit()*, described below). However, we can have the exit handler check whether a global flag is set before it performs its actions, and disable the exit handler by clearing the flag.

Exit handlers registered with *atexit()* suffer a couple of limitations. The first is that when called, an exit handler doesn't know what status was passed to *exit()*. Occasionally, knowing the status could be useful; for example, we may like to perform different actions depending on whether the process is exiting successfully or unsuccessfully. The second limitation is that we can't specify an argument to the exit handler when it is called. Such a facility could be useful to define an exit handler that performs different actions depending on its argument, or to register a function multiple times, each time with a different argument.

To address these limitations, *glibc* provides a (nonstandard) alternative method of registering exit handlers: *on_exit()*.

```
#define _BSD_SOURCE          /* Or: #define _SVID_SOURCE */
#include <stdlib.h>

int on_exit(void (*func)(int, void *), void *arg);
```

Returns 0 on success, or nonzero on error

The *func* argument of *on_exit()* is a pointer to a function of the following type:

```
void
func(int status, void *arg)
{
    /* Perform cleanup actions */
}
```

When called, *func()* is passed two arguments: the *status* argument supplied to *exit()*, and a copy of the *arg* argument supplied to *on_exit()* at the time the function was registered. Although defined as a pointer type, *arg* is open to programmer-defined interpretation. It could be used as a pointer to some structure; equally, through judicious use of casting, it could be treated as an integer or other scalar type.

Like *atexit()*, *on_exit()* returns a nonzero value (not necessarily –1) on error.

As with *atexit()*, multiple exit handlers can be registered with *on_exit()*. Functions registered using *atexit()* and *on_exit()* are placed on the same list. If both methods are used in the same program, then the exit handlers are called in reverse order of their registration using the two methods.

Although more flexible than *atexit()*, *on_exit()* should be avoided in programs intended to be portable, since it is not covered by any standards and is available on few other UNIX implementations.

Example program

Listing 25-1 demonstrates the use of *atexit()* and *on_exit()* to register exit handlers. When we run this program, we see the following output:

```
$ ./exit_handlers
on_exit function called: status=2, arg=20
atexit function 2 called
atexit function 1 called
on_exit function called: status=2, arg=10
```

Listing 25-1: Using exit handlers

procexec/exit_handlers.c

```
#define _BSD_SOURCE     /* Get on_exit() declaration from <stdlib.h> */
#include <stdlib.h>
#include "tlpi_hdr.h"

static void
atexitFunc1(void)
{
    printf("atexit function 1 called\n");
}

static void
atexitFunc2(void)
{
    printf("atexit function 2 called\n");
}

static void
onexitFunc(int exitStatus, void *arg)
{
    printf("on_exit function called: status=%d, arg=%ld\n",
                exitStatus, (long) arg);
}

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

```
{
    if (on_exit(onexitFunc, (void *) 10) != 0)
        fatal("on_exit 1");
    if (atexit(atexitFunc1) != 0)
        fatal("atexit 1");
    if (atexit(atexitFunc2) != 0)
        fatal("atexit 2");
    if (on_exit(onexitFunc, (void *) 20) != 0)
        fatal("on_exit 2");

    exit(2);
}
```

procexec/exit_handlers.c

25.4 Interactions Between *fork()*, *stdio* Buffers, and *_exit()*

The output yielded by the program in Listing 25-2 demonstrates a phenomenon that is at first puzzling. When we run this program with standard output directed to the terminal, we see the expected result:

```
$ ./fork_stdio_buf
Hello world
Ciao
```

However, when we redirect standard output to a file, we see the following:

```
$ ./fork_stdio_buf > a
$ cat a
Ciao
Hello world
Hello world
```

In the above output, we see two strange things: the line written by *printf()* appears twice, and the output of *write()* precedes that of *printf()*.

Listing 25-2: Interaction of *fork()* and *stdio* buffering

procexec/fork_stdio_buf.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Hello world\n");
    write(STDOUT_FILENO, "Ciao\n", 5);

    if (fork() == -1)
        errExit("fork");

    /* Both child and parent continue execution here */

    exit(EXIT_SUCCESS);
}
```

procexec/fork_stdio_buf.c

To understand why the message written with *printf()* appears twice, recall that the *stdio* buffers are maintained in a process's user-space memory (refer to Section 13.2). Therefore, these buffers are duplicated in the child by *fork()*. When standard output is directed to a terminal, it is line-buffered by default, with the result that the newline-terminated string written by *printf()* appears immediately. However, when standard output is directed to a file, it is block-buffered by default. Thus, in our example, the string written by *printf()* is still in the parent's *stdio* buffer at the time of the *fork()*, and this string is duplicated in the child. When the parent and the child later call *exit()*, they both flush their copies of the *stdio* buffers, resulting in duplicate output.

We can prevent this duplicated output from occurring in one of the following ways:

- As a specific solution to the *stdio* buffering issue, we can use *fflush()* to flush the *stdio* buffer prior to a *fork()* call. Alternatively, we could use *setvbuf()* or *setbuf()* to disable buffering on the *stdio* stream.
- Instead of calling *exit()*, the child can call *_exit()*, so that it doesn't flush *stdio* buffers. This technique exemplifies a more general principle: in an application that creates child processes, typically only one of the processes (most often the parent) should terminate via *exit()*, while the other processes should terminate via *_exit()*. This ensures that only one process calls exit handlers and flushes *stdio* buffers, which is usually desirable.

> Other approaches that allow both the parent and child to call *exit()* are possible (and sometimes necessary). For example, it may be possible to design exit handlers so that they operate correctly even if called from multiple processes, or to have the application install exit handlers only after the call to *fork()*. Furthermore, sometimes we may actually want all processes to flush their *stdio* buffers after a *fork()*. In this case, we may choose to terminate the processes using *exit()*, or use explicit calls to *fflush()* in each process, as appropriate.

The output of the *write()* in the program in Listing 25-2 doesn't appear twice, because *write()* transfers data directly to a kernel buffer, and this buffer is not duplicated during a *fork()*.

By now, the reason for the second strange aspect of the program's output when redirected to a file should be clear. The output of *write()* appears before that from *printf()* because the output of *write()* is immediately transferred to the kernel buffer cache, while the output from *printf()* is transferred only when the *stdio* buffers are flushed by the call to *exit()*. (In general, care is required when mixing *stdio* functions and system calls to perform I/O on the same file, as described in Section 13.7.)

25.5 Summary

A process can terminate either abnormally or normally. Abnormal termination occurs on delivery of certain signals, some of which also cause the process to produce a core dump file.

Normal termination is accomplished by calling *_exit()* or, more usually, *exit()*, which is layered on top of *_exit()*. Both *_exit()* and *exit()* take an integer argument whose least significant 8 bits define the termination status of the process. By convention, a status of 0 is used to indicate successful termination, and a nonzero status indicates unsuccessful termination.

As part of both normal and abnormal process termination, the kernel performs various cleanup steps. Terminating a process normally by calling *exit()* additionally causes exit handlers registered using *atexit()* and *on_exit()* to be called (in reverse order of registration), and causes *stdio* buffers to be flushed.

Further information

Refer to the sources of further information listed in Section 24.6.

25.6 Exercise

25-1. If a child process makes the call *exit(−1)*, what exit status (as returned by `WEXITSTATUS()`) will be seen by the parent?