27

PROGRAM EXECUTION

This chapter follows from our discussion of process creation and termination in the previous chapters. We now look at how a process can use the *execve()* system call to replace the program that it is running by a completely new program. We then show how to implement the *system()* function, which allows its caller to execute an arbitrary shell command.

27.1 Executing a New Program: *execve()*

The *execve()* system call loads a new program into a process's memory. During this operation, the old program is discarded, and the process's stack, data, and heap are replaced by those of the new program. After executing various C library run-time startup code and program initialization code (e.g., C++ static constructors or C functions declared with the *gcc* `constructor` attribute described in Section 42.4), the new program commences execution at its *main()* function.

The most frequent use of *execve()* is in the child produced by a *fork()*, although it is also occasionally used in applications without a preceding *fork()*.

Various library functions, all with names beginning with *exec*, are layered on top of the *execve()* system call. Each of these functions provides a different interface to the same functionality. The loading of a new program by any of these calls is commonly referred to as an *exec* operation, or simply by the notation *exec()*. We begin with a description of *execve()* and then describe the library functions.

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[], char *const envp[]);
                                        Never returns on success; returns –1 on error
```

The *pathname* argument contains the pathname of the new program to be loaded into the process's memory. This pathname can be absolute (indicated by an initial /) or relative to the current working directory of the calling process.

The *argv* argument specifies the command-line arguments to be passed to the new program. This array corresponds to, and has the same form as, the second (*argv*) argument to a C *main()* function; it is a NULL-terminated list of pointers to character strings. The value supplied for *argv[0]* corresponds to the command name. Typically, this value is the same as the basename (i.e., the final component) of *pathname*.

The final argument, *envp*, specifies the environment list for the new program. The *envp* argument corresponds to the *environ* array of the new program; it is a NULL-terminated list of pointers to character strings of the form *name=value* (Section 6.7).

> The Linux-specific /proc/*PID*/exe file is a symbolic link containing the absolute pathname of the executable file being run by the corresponding process.

After an *execve()*, the process ID of the process remains the same, because the same process continues to exist. A few other process attributes also remain unchanged, as described in Section 28.4.

If the set-user-ID (set-group-ID) permission bit of the program file specified by *pathname* is set, then, when the file is execed, the effective user (group) ID of the process is changed to be the same as the owner (group) of the program file. This is a mechanism for temporarily granting privileges to users while running a specific program (see Section 9.3).

After optionally changing the effective IDs, and regardless of whether they were changed, an *execve()* copies the value of the process's effective user ID into its saved set-user-ID, and copies the value of the process's effective group ID into its saved set-group-ID.

Since it replaces the program that called it, a successful *execve()* never returns. We never need to check the return value from *execve()*; it will always be –1. The very fact that it returned informs us that an error occurred, and, as usual, we can use *errno* to determine the cause. Among the errors that may be returned in *errno* are the following:

EACCES
: The *pathname* argument doesn't refer to a regular file, the file doesn't have execute permission enabled, or one of the directory components of *pathname* is not searchable (i.e., execute permission is denied on the directory). Alternatively, the file resides on a file system that was mounted with the MS_NOEXEC flag (Section 14.8.1).

ENOENT
: The file referred to by *pathname* doesn't exist.

ENOEXEC
: The file referred to by *pathname* is marked as being executable, but it is not in a recognizable executable format. Possibly, it is a script that doesn't begin with a line (starting with the characters #!) specifying a script interpreter.

ETXTBSY
: The file referred to by *pathname* is open for writing by another process (Section 4.3.2).

E2BIG
: The total space required by the argument list and environment list exceeds the allowed maximum.

The errors listed above may also be generated if any of these conditions apply to the interpreter file defined to execute a script (refer to Section 27.3) or to the ELF interpreter being used to execute the program.

> The Executable and Linking Format (ELF) is a widely implemented specification describing the layout of executable files. Normally, during an exec, a process image is constructed using the segments of the executable file (Section 6.3). However, the ELF specification also allows for an executable file to define an interpreter (the PT_INTERP ELF program header element) to be used to execute the program. If an interpreter is defined, the kernel constructs the process image from the segments of the specified interpreter executable file. It is then the responsibility of the interpreter to load and execute the program. We say a little more about the ELF interpreter in Chapter 41 and provide some pointers to further information in that chapter.

Example program

Listing 27-1 demonstrates the use of *execve()*. This program creates an argument list and an environment for a new program, and then calls *execve()*, using its command-line argument (*argv[1]*) as the pathname to be executed.

Listing 27-2 shows a program that is designed to be executed by the program in Listing 27-1. This program simply displays its command-line arguments and environment list (the latter is accessed using the global *environ* variable, as described in Section 6.7).

The following shell session demonstrates the use of the programs in Listing 27-1 and Listing 27-2 (in this example, a relative pathname is used to specify the program to be execed):

```
$ ./t_execve ./envargs
argv[0] = envargs
argv[1] = hello world
argv[2] = goodbye
environ: GREET=salut
environ: BYE=adieu
```

All of the output is printed by envargs

Listing 27-1: Using *execve()* to execute a new program

procexec/t_execve.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *argVec[10];           /* Larger than required */
    char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    argVec[0] = strrchr(argv[1], '/');      /* Get basename from argv[1] */
    if (argVec[0] != NULL)
        argVec[0]++;
    else
        argVec[0] = argv[1];
    argVec[1] = "hello world";
    argVec[2] = "goodbye";
    argVec[3] = NULL;           /* List must be NULL-terminated */

    execve(argv[1], argVec, envVec);
    errExit("execve");          /* If we get here, something went wrong */
}
```

procexec/t_execve.c

Listing 27-2: Display argument list and environment

procexec/envargs.c

```
#include "tlpi_hdr.h"

extern char **environ;

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

    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);

    for (ep = environ; *ep != NULL; ep++)
        printf("environ: %s\n", *ep);

    exit(EXIT_SUCCESS);
}
```

procexec/envargs.c

27.2 The *exec()* Library Functions

The library functions described in this section provide alternative APIs for performing an *exec()*. All of these functions are layered on top of *execve()*, and they differ from one another and from *execve()* only in the way in which the program name, argument list, and environment of the new program are specified.

```
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...
                /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
                /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
                /* , (char *) NULL */);
```

None of the above returns on success; all return −1 on error

The final letters in the names of these functions provide a clue to the differences between them. These differences are summarized in Table 27-1 and detailed in the following list:

- Most of the *exec()* functions expect a pathname as the specification of the new program to be loaded. However, *execlp()* and *execvp()* allow the program to be specified using just a filename. The filename is sought in the list of directories specified in the PATH environment variable (explained in more detail below). This is the kind of searching that the shell performs when given a command name. To indicate this difference in operation, the names of these functions contain the letter *p* (for PATH). The PATH variable is not used if the filename contains a slash (/), in which case it is treated as a relative or absolute pathname.
- Instead of using an array to specify the *argv* list for the new program, *execle()*, *execlp()*, and *execl()* require the programmer to specify the arguments as a list of strings within the call. The first of these arguments corresponds to *argv[0]* in the *main* function of the new program, and is thus typically the same as the *filename* argument or the basename component of the *pathname* argument. A NULL pointer must terminate the argument list, so that these calls can locate the end of the list. (This requirement is indicated by the commented *(char *) NULL* in the above prototypes; for a discussion of why the cast is required before the NULL, see Appendix C.) The names of these functions contain the letter *l* (for *list*) to distinguish them from those functions requiring the argument list as a NULL-terminated array. The names of the functions that require the argument list as an array (*execve()*, *execvp()*, and *execv()*) contain the letter *v* (for *vector*).

- The *execve()* and *execle()* functions allow the programmer to explicitly specify the environment for the new program using *envp*, a NULL-terminated array of pointers to character strings. The names of these functions end with the letter *e* (for *environment*) to indicate this fact. All of the other *exec()* functions use the caller's existing environment (i.e., the contents of *environ*) as the environment for the new program.

> Version 2.11 of *glibc* added a nonstandard function, *execvpe(file, argv, envp)*. This function is like *execvp()*, but instead of taking the environment for the new program from *environ*, the caller specifies the new environment via the *envp* argument (like *execve()* and *execle()*).

In the next few pages, we demonstrate the use of some of these *exec()* variants.

Table 27-1: Summary of differences between the *exec()* functions

Function	Specification of program file *(–, p)*	Specification of arguments *(v, l)*	Source of environment *(e, –)*
execve()	pathname	array	*envp* argument
execle()	pathname	list	*envp* argument
execlp()	filename + PATH	list	caller's *environ*
execvp()	filename + PATH	array	caller's *environ*
execv()	pathname	array	caller's *environ*
execl()	pathname	list	caller's *environ*

27.2.1 The PATH Environment Variable

The *execvp()* and *execlp()* functions allow us to specify just the name of the file to be executed. These functions make use of the PATH environment variable to search for the file. The value of PATH is a string consisting of colon-separated directory names called *path prefixes*. As an example, the following PATH value specifies five directories:

```
$ echo $PATH
/home/mtk/bin:/usr/local/bin:/usr/bin:/bin:.
```

The PATH value for a login shell is set by system-wide and user-specific shell startup scripts. Since a child process inherits a copy of its parent's environment variables, each process that the shell creates to execute a command inherits a copy of the shell's PATH.

The directory pathnames specified in PATH can be either absolute (commencing with an initial /) or relative. A relative pathname is interpreted with respect to the current working directory of the calling process. The current working directory can be specified using . (dot), as in the above example.

> It is also possible to specify the current working directory by including a zero-length prefix in PATH, by employing consecutive colons, an initial colon, or a trailing colon (e.g., /usr/bin:/bin:). SUSv3 declares this technique obsolete; the current working directory should be explicitly specified using . (dot).

If the PATH variable is not defined, then *execvp()* and *execlp()* assume a default path list of .:/usr/bin:/bin.

As a security measure, the superuser account (*root*) is normally set up so that the current working directory is excluded from PATH. This prevents *root* from accidentally executing a file from the current working directory (which may have been deliberately placed there by a malicious user) with the same name as a standard command or with a name that is a misspelling of a common command (e.g., *sl* instead of *ls*). In some Linux distributions, the default value for PATH also excludes the current working directory for unprivileged users. We assume such a PATH definition in all of the shell session logs shown in this book, which is why we always prefix ./ to the names of programs executed from the current working directory. (This also has the useful side effect of visually distinguishing our programs from standard commands in the shell session logs shown in this book.)

The *execvp()* and *execlp()* functions search for the filename in each of the directories named in PATH, starting from the beginning of the list and continuing until a file with the given name is successfully execed. Using the PATH environment variable in this way is useful if we don't know the run-time location of an executable file or don't want to create a hard-coded dependency on that location.

The use of *execvp()* and *execlp()* in set-user-ID or set-group-ID programs should be avoided, or at least approached with great caution. In particular, the PATH environment variable should be carefully controlled to prevent the execing of a malicious program. In practice, this means that the application should override any previously defined PATH value with a known-secure directory list.

Listing 27-3 provides an example of the use of *execlp()*. The following shell session log demonstrates the use of this program to invoke the *echo* command (/bin/echo):

```
$ which echo
/bin/echo
$ ls -l /bin/echo
-rwxr-xr-x    1 root      15428 Mar 19 21:28 /bin/echo
$ echo $PATH                        Show contents of PATH environment variable
/home/mtk/bin:/usr/local/bin:/usr/bin:/bin          /bin is in PATH
$ ./t_execlp echo                   execlp() uses PATH to successfully find echo
hello world
```

The string *hello world* that appears above was supplied as the third argument of the call to *execlp()* in the program in Listing 27-3.

We continue by redefining PATH to omit /bin, which is the directory containing the *echo* program:

```
$ PATH=/home/mtk/bin:/usr/local/bin:/usr/bin
$ ./t_execlp echo
ERROR [ENOENT No such file or directory] execlp
$ ./t_execlp /bin/echo
hello world
```

As can be seen, when we supply a filename (i.e., a string containing no slashes) to *execlp()*, the call fails, since a file named echo was not found in any of the directories listed in PATH. On the other hand, when we provide a pathname containing one or more slashes, *execlp()* ignores the contents of PATH.

Listing 27-3: Using *execlp()* to search for a filename in PATH

```
                                                          procexec/t_execlp.c
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    execlp(argv[1], argv[1], "hello world", (char *) NULL);
    errExit("execlp");          /* If we get here, something went wrong */
}
                                                          procexec/t_execlp.c
```

27.2.2 Specifying Program Arguments as a List

When we know the number of arguments for an *exec()* at the time we write a program, we can use *execle()*, *execlp()*, or *execl()* to specify the arguments as a list within the function call. This can be convenient, since it requires less code than assembling the arguments in an *argv* vector. The program in Listing 27-4 achieves the same result as the program in Listing 27-1 but using *execle()* instead of *execve()*.

Listing 27-4: Using *execle()* to specify program arguments as a list

```
                                                          procexec/t_execle.c
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };
    char *filename;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    filename = strrchr(argv[1], '/');       /* Get basename from argv[1] */
    if (filename != NULL)
        filename++;
    else
        filename = argv[1];

    execle(argv[1], filename, "hello world", (char *) NULL, envVec);
    errExit("execle");          /* If we get here, something went wrong */
}
                                                          procexec/t_execle.c
```

27.2.3 Passing the Caller's Environment to the New Program

The *execlp()*, *execvp()*, *execl()*, and *execv()* functions don't permit the programmer to explicitly specify an environment list; instead, the new program inherits its environment from the calling process (Section 6.7). This may, or may not, be desirable. For

security reasons, it is sometimes preferable to ensure that a program is execed with a known environment list. We consider this point further in Section 38.8.

Listing 27-5 demonstrates that the new program inherits its environment from the caller during an *execl()* call. This program first uses *putenv()* to make a change to the environment that it inherits from the shell as a result of *fork()*. Then the *printenv* program is execed to display the values of the `USER` and `SHELL` environment variables. When we run this program, we see the following:

```
$ echo $USER $SHELL           Display some of the shell's environment variables
blv /bin/bash
$ ./t_execl
Initial value of USER: blv    Copy of environment was inherited from the shell
britta                        These two lines are displayed by execed printenv
/bin/bash
```

Listing 27-5: Passing the caller's environment to the new program using *execl()*

procexec/t_execl.c

```
#include <stdlib.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Initial value of USER: %s\n", getenv("USER"));
    if (putenv("USER=britta") != 0)
        errExit("putenv");

    execl("/usr/bin/printenv", "printenv", "USER", "SHELL", (char *) NULL);
    errExit("execl");           /* If we get here, something went wrong */
}
```

procexec/t_execl.c

27.2.4 Executing a File Referred to by a Descriptor: *fexecve()*

Since version 2.3.2, *glibc* provides *fexecve()*, which behaves just like *execve()*, but specifies the file to be execed via the open file descriptor *fd*, rather than as a pathname. Using *fexecve()* is useful for applications that want to open a file, verify its contents by performing a checksum, and then execute the file.

```
#define _GNU_SOURCE
#include <unistd.h>

int fexecve(int fd, char *const argv[], char *const envp[]);
```

Doesn't return on success; returns –1 on error

Without *fexecve()*, we could *open()* and read the file to verify its contents, and then exec it. However, this would allow the possibility that, between opening the file and execing it, the file was replaced (holding an open file descriptor doesn't prevent a new file with the same name from being created), so that the content that was execed was different from the content that was checked.

27.3 Interpreter Scripts

An *interpreter* is a program that reads commands in text form and executes them. (This contrasts with a *compiler*, which translates input source code into a machine language that can then be executed on a real or virtual machine.) Examples of interpreters include the various UNIX shells and programs such as *awk*, *sed*, *perl*, *python*, and *ruby*. In addition to being able to read and execute commands interactively, interpreters usually provide a facility to read and execute commands from a text file, referred to as a *script*.

UNIX kernels allow interpreter scripts to be run in the same way as a binary program file, as long as two requirements are met. First, execute permission must be enabled for the script file. Second, the file must contain an initial line that specifies the pathname of the interpreter to be used to run the script. This line has the following form:

```
#! interpreter-path [ optional-arg ]
```

The `#!` characters must be placed at the start of the line; optionally, a space may separate these characters from the interpreter pathname. The `PATH` environment variable is *not* used in interpreting this pathname, so that an absolute pathname usually should be specified. A relative pathname is also possible, though unusual; it is interpreted relative to the current working directory of the process starting the interpreter. White space separates the interpreter pathname from an optional argument, whose purpose we explain shortly. The optional argument should not contain white-space characters.

As an example, UNIX shell scripts usually begin with the following line, which specifies that the shell is to be used to execute the script:

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

> The optional argument in the first line of the interpreter script file should not contain white space because the behavior in this case is highly implementation-dependent. On Linux, white space in *optional-arg* is not interpreted specially—all of the text from the start of the argument to the end of the line is interpreted as a single word (which is given as an argument to the script, as we describe below). Note that this treatment of spaces contrasts with the shell, where white space delimits the words of a command line.
>
> While some UNIX implementations treat white space in *optional-arg* in the same way as Linux, other implementations do not. On FreeBSD before version 6.0, multiple space-delimited optional arguments may follow *interpreter-path* (and these are passed as separate words to the script); since version 6.0, FreeBSD behaves like Linux. On Solaris 8, white-space characters terminate *optional-arg*, and any remaining text in the `#!` line is ignored.

The Linux kernel places a 127-character limit on the length of the `#!` line of a script (excluding the newline character at the end of the line). Additional characters are silently ignored.

The `#!` technique for interpreter scripts is not specified in SUSv3, but is available on most UNIX implementations.

The limit placed on the length of the #! line varies across UNIX implementations. For example, the limit is 64 characters in OpenBSD 3.1 and 1024 characters on Tru64 5.1. On some historical implementations (e.g., SunOS 4), this limit was as low as 32 characters.

Execution of interpreter scripts

Since a script doesn't contain binary machine code, when *execve()* is used to run the script, obviously something different from usual must be occurring when the script is executed. If *execve()* detects that the file it has been given commences with the 2-byte sequence #!, then it extracts the remainder of the line (the pathname and argument), and execs the interpreter file with the following list of arguments:

---

interpreter-path [*optional-arg*] *script-path arg...*

Here, *interpreter-path* and *optional-arg* are taken from the #! line of the script, *script-path* is the pathname given to *execve()*, and *arg...* is the list of any further arguments specified via the *argv* argument to *execve()* (but excluding *argv[0]*). The origin of each of the script arguments is summarized in Figure 27-1.

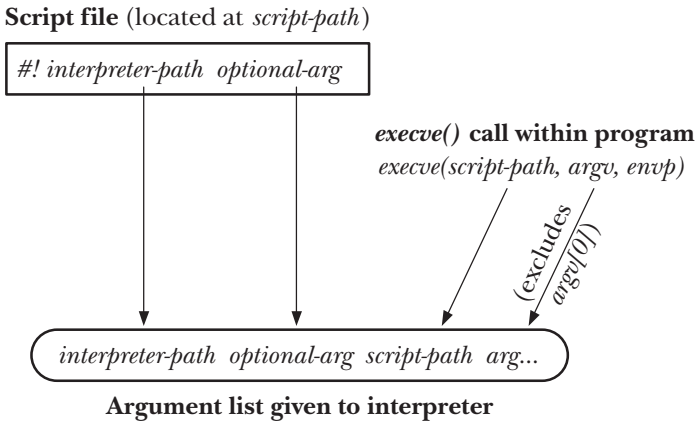

Figure 27-1: The argument list supplied to an execed script

We can demonstrate the origin of the interpreter arguments by writing a script that uses the program in Listing 6-2 (necho.c, on page 123) as an interpreter. This program simply echoes all of its command-line arguments. We then use the program in Listing 27-1 to exec the script:

```
$ cat > necho.script                Create script
#!/home/mtk/bin/necho some argument
Some junk
Type Control-D
$ chmod +x necho.script             Make script executable
$ ./t_execve necho.script           And exec the script
argv[0] = /home/mtk/bin/necho       First 3 arguments are generated by kernel
argv[1] = some argument             Script argument is treated as a single word
argv[2] = necho.script              This is the script path
argv[3] = hello world               This was argVec[1] given to execve()
argv[4] = goodbye                   And this was argVec[2]
```

In this example, our "interpreter" (`necho`) ignores the contents of its script file (`necho.script`), and the second line of the script (*Some junk*) has no effect on its execution.

> The Linux 2.2 kernel passes only the basename part of the *interpreter-path* as the first argument when invoking a script. Consequently, on Linux 2.2, the line displaying *argv[0]* would show just the value *echo*.

Most UNIX shells and interpreters treat the `#` character as the start of a comment. Thus, these interpreters ignore the initial `#!` line when interpreting the script.

Using the script *optional-arg*

One use of the *optional-arg* in a script's initial `#!` line is to specify command-line options for the interpreter. This feature is useful with certain interpreters, such as *awk*.

> The *awk* interpreter has been part of the UNIX system since the late 1970s. The *awk* language is described in a number of books, including one by its creators [Aho et al., 1988], whose initials gave the language its name. Its forte is rapid prototyping of text-processing applications. In its design—a weakly typed language, with a rich set of text-handling primitives, and a syntax based on C—*awk* is the ancestor of many widely used contemporary scripting languages, such as JavaScript and PHP.

A script can be supplied to *awk* in two different ways. The default is to provide the script as the first command-line argument to *awk*:

```
$ awk 'script' input-file...
```

Alternatively, an *awk* script can reside inside a file, as in the following *awk* script, which prints out the length of the longest line of its input:

```
$ cat longest_line.awk
#!/usr/bin/awk
length > max { max = length; }
END          { print max; }
```

Suppose that we try execing this script using the following C code:

```
execl("longest_line.awk", "longest_line.awk", "input.txt", (char *) NULL);
```

This *execl()* call in turn employs *execve()* with the following argument list to invoke *awk*:

```
/usr/bin/awk longest_line.awk input.txt
```

This *execve()* call fails, because *awk* interprets the string *longest_line.awk* as a script containing an invalid *awk* command. We need a way of informing *awk* that this argument is actually the name of a file containing the script. We can do this by adding the *-f* option as the optional argument in the script's `#!` line. This tells *awk* that the following argument is a script file:

```
#!/usr/bin/awk -f
length > max { max = length; }
END          { print max; }
```

Now, our *execl()* call results in the following argument list being used:

```
/usr/bin/awk -f longest_line.awk input.txt
```

This successfully invokes *awk* using the script `longest_line.awk` to process the file `input.txt`.

Executing scripts with *execlp()* and *execvp()*

Normally, the absence of a `#!` line at the start of a script causes the *exec()* functions to fail. However, *execlp()* and *execvp()* do things somewhat differently. Recall that these are the functions that use the `PATH` environment variable to obtain a list of directories in which to search for a file to be executed. If either of these functions finds a file that has execute permission turned on, but is not a binary executable and does not start with a `#!` line, then they exec the shell to interpret the file. On Linux, this means that such files are treated as though they started with a line containing the string `#!/bin/sh`.

27.4 File Descriptors and *exec()*

By default, all file descriptors opened by a program that calls *exec()* remain open across the *exec()* and are available for use by the new program. This is frequently useful, because the calling program may open files on particular descriptors, and these files are automatically available to the new program, without it needing to know the names of, or open, the files.

The shell takes advantage of this feature to handle I/O redirection for the programs that it executes. For example, suppose we enter the following shell command:

```
$ ls /tmp > dir.txt
```

The shell performs the following steps to execute this command:

1. A *fork()* is performed to create a child process that is also running a copy of the shell (and thus has a copy of the command).
2. The child shell opens `dir.txt` for output using file descriptor 1 (standard output). This can be done in either of the following ways:
   a) The child shell closes descriptor 1 (`STDOUT_FILENO`) and then opens the file `dir.txt`. Since *open()* always uses the lowest available file descriptor, and standard input (descriptor 0) remains open, the file will be opened on descriptor 1.
   b) The shell opens `dir.txt`, obtaining a new file descriptor. Then, if that file descriptor is not standard output, the shell uses *dup2()* to force standard output to be a duplicate of the new descriptor and closes the new descriptor, since it is no longer required. (This method is safer than the preceding

method, since it doesn't rely on lower-numbered descriptors being open.) The code sequence is something like the following:

```
fd = open("dir.txt", O_WRONLY | O_CREAT,
          S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
           /* rw-rw-rw- */
if (fd != STDOUT_FILENO) {
    dup2(fd, STDOUT_FILENO);
    close(fd);
}
```

3. The child shell execs the *ls* program. The *ls* program writes its output to standard output, which is the file `dir.txt`.

> The explanation given here of how the shell performs I/O redirections simplifies some points. In particular, certain commands—so-called shell *built-in* commands—are executed directly by the shell, without performing a *fork()* or an *exec()*. Such commands must be treated somewhat differently for the purposes of I/O redirection.
>
> A shell command is implemented as a built-in for either of two reasons: efficiency and to obtain side effects within the shell. Some frequently used commands—such as *pwd*, *echo*, and *test*—are sufficiently simple that it is a worthwhile efficiency to implement them inside the shell. Other commands are implemented within the shell so that they have side effects on the shell itself—that is, they change information stored by the shell, or modify attributes of or affect the execution of the shell process. For example, the *cd* command must change the working directory of the shell itself, and so can't be executed within a separate process. Other examples of commands that are built in for their side effects include *exec*, *exit*, *read*, *set*, *source*, *ulimit*, *umask*, *wait*, and the shell job-control commands (*jobs*, *fg*, and *bg*). The full set of built-in commands understood by a shell is documented in the shell's manual page.

The close-on-exec flag (`FD_CLOEXEC`)

Sometimes, it may be desirable to ensure that certain file descriptors are closed before an *exec()*. In particular, if we *exec()* an unknown program (i.e., one that we did not write) from a privileged process, or a program that doesn't need descriptors for files we have already opened, then it is secure programming practice to ensure that all unnecessary file descriptors are closed before the new program is loaded. We could do this by calling *close()* on all such descriptors, but this suffers the following limitations:

- The file descriptor may have been opened by a library function. This function has no mechanism to force the main program to close the file descriptor before the *exec()* is performed. (As a general principle, library functions should always set the close-on-exec flag, using the technique described below, for any files that they open.)

- If the *exec()* call fails for some reason, we may want to keep the file descriptors open. If they are already closed, it may be difficult, or impossible, to reopen them so that they refer to the same files.

For these reasons, the kernel provides a close-on-exec flag for each file descriptor. If this flag is set, then the file descriptor is automatically closed during a successful *exec()*, but left open if the *exec()* fails. The close-on-exec flag for a file descriptor can be accessed using the *fcntl()* system call (Section 5.2). The *fcntl()* `F_GETFD` operation retrieves a copy of the file descriptor flags:

```
int flags;

flags = fcntl(fd, F_GETFD);
if (flags == -1)
    errExit("fcntl");
```

After retrieving these flags, we can modify the `FD_CLOEXEC` bit and use a second *fcntl()* call specifying `F_SETFD` to update the flags:

```
flags |= FD_CLOEXEC;
if (fcntl(fd, F_SETFD, flags) == -1)
    errExit("fcntl");
```

> `FD_CLOEXEC` is actually the only bit used in the file descriptor flags. This bit corresponds to the value 1. In older programs, we may sometimes see the close-on-exec flag set using just the call *fcntl(fd, F_SETFD, 1)*, relying on the fact that there are no other bits that can be affected by this operation. Theoretically, this may not always be so (in the future, some UNIX system might implement additional flag bits), so we should use the technique shown in the main text.
>
> Many UNIX implementations, including Linux, also allow the close-on-exec flag to be modified using two unstandardized *ioctl()* calls: *ioctl(fd, FIOCLEX)* to set the close-on-exec flag for *fd*, and *ioctl(fd, FIONCLEX)* to clear the flag.

When *dup()*, *dup2()*, or *fcntl()* is used to create a duplicate of a file descriptor, the close-on-exec flag is always cleared for the duplicate descriptor. (This behavior is historical and an SUSv3 requirement.)

Listing 27-6 demonstrates the manipulation of the close-on-exec flag. Depending on the presence of a command-line argument (any string), this program first sets the close-on-exec flag for standard output and then execs the *ls* program. Here is what we see when we run the program:

```
$ ./closeonexec                    Exec ls without closing standard output
-rwxr-xr-x   1 mtk    users    28098 Jun 15 13:59 closeonexec
$ ./closeonexec n                  Sets close-on-exec flag for standard output
ls: write error: Bad file descriptor
```

In the second run shown above, *ls* detects that its standard output is closed and prints an error message on standard error.

Listing 27-6: Setting the close-on-exec flag for a file descriptor

procexec/closeonexec.c

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

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

    if (argc > 1) {
        flags = fcntl(STDOUT_FILENO, F_GETFD);              /* Fetch flags */
        if (flags == -1)
            errExit("fcntl - F_GETFD");

        flags |= FD_CLOEXEC;                    /* Turn on FD_CLOEXEC */

        if (fcntl(STDOUT_FILENO, F_SETFD, flags) == -1)     /* Update flags */
            errExit("fcntl - F_SETFD");
    }

    execlp("ls", "ls", "-l", argv[0], (char *) NULL);
    errExit("execlp");
}
```

procexec/closeonexec.c

27.5 Signals and *exec()*

During an *exec()*, the text of the existing process is discarded. This text may include signal handlers established by the calling program. Because the handlers disappear, the kernel resets the dispositions of all handled signals to SIG_DFL. The dispositions of all other signals (i.e., those with dispositions of SIG_IGN or SIG_DFL) are left unchanged by an *exec()*. This behavior is required by SUSv3.

SUSv3 makes a special case for an ignored SIGCHLD signal. (We noted in Section 26.3.3 that ignoring SIGCHLD prevents the creation of zombies.) SUSv3 leaves it unspecified whether an ignored SIGCHLD remains ignored across an *exec()* or its disposition is reset to SIG_DFL. Linux does the former, but some other UNIX implementations (e.g., Solaris) do the latter. This implies that, in programs that ignore SIGCHLD, for maximum portability, we should perform a *signal(SIGCHLD, SIG_DFL)* call prior to an *exec()*, and ensure that we don't write programs that rely on the initial disposition of SIGCHLD being anything other than SIG_DFL.

The destruction of the old program's data, heap, and stack also means that any alternate signal stack established by a call to *sigaltstack()* (Section 21.3) is lost. Since an alternate signal stack is not preserved across an *exec()*, the SA_ONSTACK bit is also cleared for all signals.

During an *exec()*, the process signal mask and set of pending signals are both preserved. This feature allows us to block and queue signals for the newly execed program. However, SUSv3 notes that many existing applications wrongly assume that they are started with the disposition of certain signals set to SIG_DFL or that these signals are unblocked. (In particular, the C standards provide a much weaker

specification of signals, which doesn't specify signal blocking; therefore, C programs written on non-UNIX systems won't know to unblock signals.) For this reason, SUSv3 recommends that signals should not be blocked or ignored across an *exec()* of an arbitrary program. Here, "arbitrary" means a program that we did not write. It is acceptable to block or ignore signals when execing a program we have written or one with known behavior with respect to signals.

27.6 Executing a Shell Command: *system()*

The *system()* function allows the calling program to execute an arbitrary shell command. In this section, we describe the operation of *system()*, and in the next section we show how *system()* can be implemented using *fork()*, *exec()*, *wait()*, and *exit()*.

> In Section 44.5, we look at the *popen()* and *pclose()* functions, which can also be used to execute a shell command, but allow the calling program to either read the output of the command or to send input to the command.

```
#include <stdlib.h>

int system(const char *command);
```

See main text for a description of return value

The *system()* function creates a child process that invokes a shell to execute *command*. Here is an example of a call to *system()*:

```
system("ls | wc");
```

The principal advantages of *system()* are simplicity and convenience:

- We don't need to handle the details of calling *fork()*, *exec()*, *wait()*, and *exit()*.
- Error and signal handling are performed by *system()* on our behalf.
- Because *system()* uses the shell to execute *command*, all of the usual shell processing, substitutions, and redirections are performed on *command* before it is executed. This makes it easy to add an "execute a shell command" feature to an application. (Many interactive applications provide such a feature in the form of a `!` command.)

The main cost of *system()* is inefficiency. Executing a command using *system()* requires the creation of at least two processes—one for the shell and one or more for the command(s) it executes—each of which performs an *exec()*. If efficiency or speed is a requirement, it is preferable to use explicit *fork()* and *exec()* calls to execute the desired program.

The return value of *system()* is as follows:

- If *command* is a `NULL` pointer, then *system()* returns a nonzero value if a shell is available, and 0 if no shell is available. This case arises out of the C programming language standards, which are not tied to any operating system, so a shell might not be available if *system()* is running on a non-UNIX system. Furthermore, even though all UNIX implementations have a shell, this shell might not

be available if the program called *chroot()* before calling *system()*. If *command* is non-NULL, then the return value for *system()* is determined according to the remaining rules in this list.

- If a child process could not be created or its termination status could not be retrieved, then *system()* returns −1.
- If a shell could not be execed in the child process, then *system()* returns a value as though the child shell had terminated with the call *_exit(127)*.
- If all system calls succeed, then *system()* returns the termination status of the child shell used to execute *command*. (The termination status of a shell is the termination status of the last command it executes.)

> It is impossible (using the value returned by *system()*) to distinguish the case where *system()* fails to exec a shell from the case where the shell exits with the status 127 (the latter possibility can occur if the shell could not find a program with the given name to exec).

In the last two cases, the value returned by *system()* is a *wait status* of the same form returned by *waitpid()*. This means we should use the functions described in Section 26.1.3 to dissect this value, and we can display the value using our *printWaitStatus()* function (Listing 26-2, on page 546).

Example program

Listing 27-7 demonstrates the use of *system()*. This program executes a loop that reads a command string, executes it using *system()*, and then analyzes and displays the value returned by *system()*. Here is a sample run:

```
$ ./t_system
Command: whoami
mtk
system() returned: status=0x0000 (0,0)
child exited, status=0
Command: ls | grep XYZ                  Shell terminates with the status of...
system() returned: status=0x0100 (1,0)  its last command (grep), which...
child exited, status=1                  found no match, and so did an exit(1)
Command: exit 127
system() returned: status=0x7f00 (127,0)
(Probably) could not invoke shell       Actually, not true in this case
Command: sleep 100
Type Control-Z to suspend foreground process group
[1]+  Stopped           ./t_system
$ ps | grep sleep                       Find PID of sleep
29361 pts/6    00:00:00 sleep
$ kill 29361                            And send a signal to terminate it
$ fg                                    Bring t_system back into foreground
./t_system
system() returned: status=0x000f (0,15)
child killed by signal 15 (Terminated)
Command: ^D$                            Type Control-D to terminate program
```

Listing 27-7: Executing shell commands with *system()*

procexec/t_system.c

```
#include <sys/wait.h>
#include "print_wait_status.h"
#include "tlpi_hdr.h"

#define MAX_CMD_LEN 200

int
main(int argc, char *argv[])
{
    char str[MAX_CMD_LEN];      /* Command to be executed by system() */
    int status;                 /* Status return from system() */

    for (;;) {                  /* Read and execute a shell command */
        printf("Command: ");
        fflush(stdout);
        if (fgets(str, MAX_CMD_LEN, stdin) == NULL)
            break;              /* end-of-file */

        status = system(str);
        printf("system() returned: status=0x%04x (%d,%d)\n",
                (unsigned int) status, status >> 8, status & 0xff);

        if (status == -1) {
            errExit("system");
        } else {
            if (WIFEXITED(status) && WEXITSTATUS(status) == 127)
                printf("(Probably) could not invoke shell\n");
            else                /* Shell successfully executed command */
                printWaitStatus(NULL, status);
        }
    }

    exit(EXIT_SUCCESS);
}
```

procexec/t_system.c

Avoid using *system()* in set-user-ID and set-group-ID programs

Set-user-ID and set-group-ID programs should never use *system()* while operating under the program's privileged identifier. Even when such programs don't allow the user to specify the text of the command to be executed, the shell's reliance on various environment variables to control its operation means that the use of *system()* inevitably opens the door for a system security breach.

For example, in older Bourne shells, the `IFS` environment variable, which defined the internal field separator used to break a command line into separate words, was the source of a number of successful system break-ins. If we defined `IFS` to have the value *a*, then the shell would treat the command string *shar* as the word *sh* followed by the argument *r*, and invoke another shell to execute the script file named `r` in the current working directory, instead of the intended purpose (executing a command named *shar*). This particular security hole was fixed by applying `IFS`

only to the words produced by shell expansions. In addition, modern shells reset IFS (to a string consisting of the three characters space, tab, and newline) on shell startup to ensure that scripts behave consistently if they inherit a strange IFS value. As a further security measure, *bash* reverts to the real user (group) ID when invoked from a set-user-ID (set-group-ID) program.

Secure programs that need to spawn another program should use *fork()* and one of the *exec()* functions—other than *execlp()* or *execvp()*—directly.

27.7 Implementing *system()*

In this section, we explain how to implement *system()*. We begin with a simple implementation, explain what pieces are missing from that implementation, and then present a complete implementation.

A simple implementation of *system()*

The *-c* option of the *sh* command provides an easy way to execute a string containing arbitrary shell commands:

```
$ sh -c "ls | wc"
     38      38     444
```

Thus, to implement *system()*, we need to use *fork()* to create a child that then does an *execl()* with arguments corresponding to the above *sh* command:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

To collect the status of the child created by *system()*, we use a *waitpid()* call that specifies the child's process ID. (Using *wait()* would not suffice, because *wait()* waits for any child, which could accidentally collect the status of some other child created by the calling process.) A simple, and incomplete, implementation of *system()* is shown in Listing 27-8.

Listing 27-8: An implementation of *system()* that excludes signal handling

procexec/simple_system.c

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

int
system(char *command)
{
    int status;
    pid_t childPid;

    switch (childPid = fork()) {
    case -1: /* Error */
        return -1;
```

```
    case 0: /* Child */
        execl("/bin/sh", "sh", "-c", command, (char *) NULL);
        _exit(127);                     /* Failed exec */

    default: /* Parent */
        if (waitpid(childPid, &status, 0) == -1)
            return -1;
        else
            return status;
    }
}
```

procexec/simple_system.c

Treating signals correctly inside *system()*

What adds complexity to the implementation of *system()* is the correct treatment with signals.

The first signal to consider is SIGCHLD. Suppose that the program calling *system()* is also directly creating children, and has established a handler for SIGCHLD that performs its own *wait()*. In this situation, when a SIGCHLD signal is generated by the termination of the child created by *system()*, it is possible that the signal handler of the main program will be invoked—and collect the child's status—before *system()* has a chance to call *waitpid()*. (This is an example of a race condition.) This has two undesirable consequences:

- The calling program would be deceived into thinking that one of the children that it created has terminated.
- The *system()* function would be unable to obtain the termination status of the child that it created.

Therefore, *system()* must block delivery of SIGCHLD while it is executing.

The other signals to consider are those generated by the terminal *interrupt* (usually *Control-C*) and *quit* (usually *Control-*) characters, SIGINT and SIGQUIT, respectively. Consider what is happening when we execute the following call:

```
system("sleep 20");
```

At this point, three processes are running: the process executing the calling program, a shell, and *sleep*, as shown in Figure 27-2.

> As an efficiency measure, when the string given to the *-c* option is a simple command (as opposed to a pipeline or a sequence), some shells (including *bash*) directly exec the command, rather than forking a child shell. For shells that perform such an optimization, Figure 27-2 is not strictly accurate, since there will be only two processes (the calling process and *sleep*). Nevertheless, the arguments in this section about how *system()* should handle signals still apply.

All of the processes shown in Figure 27-2 form part of the foreground process group for the terminal. (We consider process groups in detail in Section 34.2.) Therefore, when we type the *interrupt* or *quit* characters, all three processes are sent the corresponding signal. The shell ignores SIGINT and SIGQUIT while waiting for its

child. However, both the calling program and the *sleep* process would, by default, be killed by these signals.

How should the calling process and the executed command respond to these signals? SUSv3 specifies the following:

- `SIGINT` and `SIGQUIT` should be ignored in the calling process while the command is being executed.
- In the child, `SIGINT` and `SIGQUIT` should be treated as they would be if the calling process did a *fork()* and *exec()*; that is, the disposition of handled signals is reset to the default, and the disposition of other signals remains unchanged.

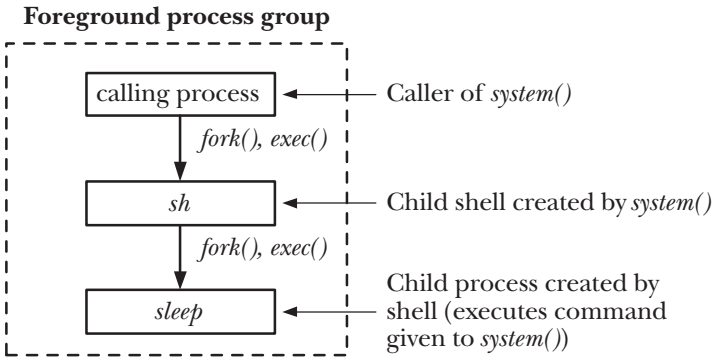

Figure 27-2: Arrangement of processes during execution of *system("sleep 20")*

Dealing with signals in the manner specified by SUSv3 is the most reasonable approach, for the following reasons:

- It would not make sense to have both processes responding to these signals, since this could lead to confusing behaviors for the user of the application.
- Similarly, it would not make sense to ignore these signals in the process executing the command while treating them according to their default dispositions in the calling process. This would allow the user to do things such as killing the calling process while the executed command was left running. It is also inconsistent with the fact that the calling process has actually given up control (i.e., is blocked in a *waitpid()* call) while the command passed to *system()* is being executed.
- The command executed by *system()* may be an interactive application, and it makes sense to have this application respond to terminal-generated signals.

SUSv3 requires the treatment of `SIGINT` and `SIGQUIT` described above, but notes that this could have an undesirable effect in a program that invisibly uses *system()* to perform some task. While the command is being executed, typing *Control-C* or *Control-* will kill only the child of *system()*, while the application (unexpectedly, to the user) continues to run. A program that uses *system()* in this way should check the termination status returned by *system()*, and take appropriate action if it detects that the command was killed by a signal.

An improved *system()* implementation

Listing 27-9 shows an implementation of *system()* conforming to the rules described above. Note the following points about this implementation:

- As noted earlier, if *command* is a `NULL` pointer, then *system()* should return nonzero if a shell is available or 0 if no shell is available. The only way to reliably determine this information is to try to execute a shell. We do this by recursively calling *system()* to execute the : shell command and checking for a return status of 0 from the recursive call ①. The : command is a shell built-in command that does nothing, but always returns a success status. We could have executed the shell command *exit 0* to achieve the same result. (Note that it isn't sufficient to use *access()* to check whether the file `/bin/sh` exists and has execute permission enabled. In a *chroot()* environment, even if the shell executable is present, it may not be executable it if it is dynamically linked and the required shared libraries are not available.)
- It is only in the parent process (the caller of *system()*) that `SIGCHLD` needs to be blocked ②, and `SIGINT` and `SIGQUIT` need to be ignored ③. However, we must perform these actions prior to the *fork()* call, because, if they were done in the parent after the *fork()*, we would create a race condition. (Suppose, for example, that the child exited before the parent had a chance to block `SIGCHLD`.) Consequently, the child must undo these changes to the signal attributes, as described shortly.
- In the parent, we ignore errors from the *sigaction()* and *sigprocmask()* calls used to manipulate signal dispositions and the signal mask ② ③ ⑨. We do this for two reasons. First, these calls are very unlikely to fail. In practice, the only thing that can realistically go wrong with these calls is an error in specifying their arguments, and such an error should be eliminated during initial debugging. Second, we assume that the caller is more interested in knowing if *fork()* or *waitpid()* failed than in knowing if these signal-manipulation calls failed. For similar reasons, we bracket the signal-manipulation calls used at the end of *system()* with code to save ⑧ and restore *errno* ⑩, so that if *fork()* or *waitpid()* fails, then the caller can determine why. If we returned −1 because these signal-manipulation calls failed, then the caller might wrongly assume that *system()* failed to execute *command*.

  > SUSv3 merely says that *system()* should return −1 if a child process could not be created or its status could not be obtained. No mention is made of a −1 return because of failures in signal-manipulation operations by *system()*.

- Error checking is not performed for signal-related system calls in the child ④ ⑤. On the one hand, there is no way of reporting such an error (the use of *_exit(127)* is reserved for reporting an error when execing the shell); on the other hand, such failures don't affect the caller of *system()*, which is a separate process.

- On return from *fork()* in the child, the disposition of SIGINT and SIGQUIT is SIG_IGN (i.e., the disposition inherited from the parent). However, as noted earlier, in the child, these signals should be treated as if the caller of *system()* did a *fork()* and an *exec()*. A *fork()* leaves the treatment of signals unchanged in the child. An *exec()* resets the dispositions of handled signals to their defaults and leaves the dispositions of other signals unchanged (Section 27.5). Therefore, if the dispositions of SIGINT and SIGQUIT in the caller were other than SIG_IGN, then the child resets the dispositions to SIG_DFL ④.

  > Some implementations of *system()* instead reset the SIGINT and SIGQUIT dispositions to those that were in effect in the caller, relying on the fact that the subsequent *execl()* will automatically reset the disposition of handled signals to their defaults. However, this could result in potentially undesirable behavior if the caller is handling either of these signals. In this case, if a signal was delivered to the child in the small time interval before the call to *execl()*, then the handler would be invoked in the child, after the signal was unblocked by *sigprocmask()*.

- If the *execl()* call in the child fails, then we use *_exit()* to terminate the process ⑥, rather than *exit()*, in order to prevent flushing of any unwritten data in the child's copy of the *stdio* buffers.
- In the parent, we must use *waitpid()* to wait specifically for the child that we created ⑦. If we used *wait()*, then we might inadvertently fetch the status of some other child created by the calling program.
- Although the implementation of *system()* doesn't require the use of a signal handler, the calling program may have established signal handlers, and one of these could interrupt a blocked call to *waitpid()*. SUSv3 explicitly requires that the wait be restarted in this case. Therefore, we use a loop to restart *waitpid()* if it fails with the error EINTR ⑦; any other error from *waitpid()* causes this loop to terminate.

Listing 27-9: Implementation of *system()*

procexec/system.c

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

int
system(const char *command)
{
    sigset_t blockMask, origMask;
    struct sigaction saIgnore, saOrigQuit, saOrigInt, saDefault;
    pid_t childPid;
    int status, savedErrno;

①   if (command == NULL)               /* Is a shell available? */
        return system(":") == 0;
```

```
    sigemptyset(&blockMask);            /* Block SIGCHLD */
    sigaddset(&blockMask, SIGCHLD);
②   sigprocmask(SIG_BLOCK, &blockMask, &origMask);

    saIgnore.sa_handler = SIG_IGN;      /* Ignore SIGINT and SIGQUIT */
    saIgnore.sa_flags = 0;
    sigemptyset(&saIgnore.sa_mask);
③   sigaction(SIGINT, &saIgnore, &saOrigInt);
    sigaction(SIGQUIT, &saIgnore, &saOrigQuit);

    switch (childPid = fork()) {
    case -1: /* fork() failed */
        status = -1;
        break;                  /* Carry on to reset signal attributes */

    case 0: /* Child: exec command */
        saDefault.sa_handler = SIG_DFL;
        saDefault.sa_flags = 0;
        sigemptyset(&saDefault.sa_mask);

④       if (saOrigInt.sa_handler != SIG_IGN)
            sigaction(SIGINT, &saDefault, NULL);
        if (saOrigQuit.sa_handler != SIG_IGN)
            sigaction(SIGQUIT, &saDefault, NULL);

⑤       sigprocmask(SIG_SETMASK, &origMask, NULL);

        execl("/bin/sh", "sh", "-c", command, (char *) NULL);
⑥       _exit(127);                 /* We could not exec the shell */

    default: /* Parent: wait for our child to terminate */
⑦       while (waitpid(childPid, &status, 0) == -1) {
            if (errno != EINTR) {       /* Error other than EINTR */
                status = -1;
                break;                  /* So exit loop */
            }
        }
        break;
    }

    /* Unblock SIGCHLD, restore dispositions of SIGINT and SIGQUIT */

⑧   savedErrno = errno;                 /* The following may change 'errno' */

⑨   sigprocmask(SIG_SETMASK, &origMask, NULL);
    sigaction(SIGINT, &saOrigInt, NULL);
    sigaction(SIGQUIT, &saOrigQuit, NULL);

⑩   errno = savedErrno;

    return status;
}
```

procexec/system.c

Further details on *system()*

Portable applications should ensure that *system()* is not called with the disposition of SIGCHLD set to SIG_IGN, because it is impossible for the *waitpid()* call to obtain the status of the child in this case. (Ignoring SIGCHLD causes the status of a child process to be immediately discarded, as described in Section 26.3.3.)

On some UNIX implementations, *system()* handles the case that it is called with the disposition of SIGCHLD set to SIG_IGN by temporarily setting the disposition of SIGCHLD to SIG_DFL. This is workable, as long as the UNIX implementation is one of those that (unlike Linux) reaps existing zombie children when the disposition of SIGCHLD is reset to SIG_IGN. (If the implementation doesn't do this, then implementing *system()* in this way would have the negative consequence that if another child that was created by the caller terminated during the execution of *system()*, it would become a zombie that might never be reaped.)

On some UNIX implementations (notably Solaris), /bin/sh is not a standard POSIX shell. If we want to ensure that we exec a standard shell, then we must use the *confstr()* library function to obtain the value of the _CS_PATH configuration variable. This value is a PATH-style list of directories containing the standard system utilities. We can assign this list to PATH, and then use *execlp()* to exec the standard shell as follows:

```
char path[PATH_MAX];

if (confstr(_CS_PATH, path, PATH_MAX) == 0)
    _exit(127);
if (setenv("PATH", path, 1) == -1)
    _exit(127);
execlp("sh", "sh", "-c", command, (char *) NULL);
_exit(127);
```

27.8 Summary

Using *execve()*, a process can replace the program that it is currently running by a new program. Arguments to the *execve()* call allow the specification of the argument list (*argv*) and environment list for the new program. Various similarly named library functions are layered on top of *execve()* and provide different interfaces to the same functionality.

All of the *exec()* functions can be used to load a binary executable file or to execute an interpreter script. When a process execs a script, the script's interpreter program replaces the program currently being executed by the process. The script's interpreter is normally identified by an initial line (starting with the characters #!) in the script that specifies the pathname of the interpreter. If no such line is present, then the script is executable only via *execlp()* or *execvp()*, and these functions exec the shell as the script interpreter.

We showed how *fork()*, *exec()*, *exit()*, and *wait()* can be combined to implement the *system()* function, which can be used to execute an arbitrary shell command.

Further information

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

27.9 Exercises

27-1. The final command in the following shell session uses the program in Listing 27-3 to exec the program *xyz*. What happens?

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:./dir1:./dir2
$ ls -l dir1
total 8
-rw-r--r--   1 mtk      users        7860 Jun 13 11:55 xyz
$ ls -l dir2
total 28
-rwxr-xr-x   1 mtk      users       27452 Jun 13 11:55 xyz
$ ./t_execlp xyz
```

27-2. Use *execve()* to implement *execlp()*. You will need to use the *stdarg(3)* API to handle the variable-length argument list supplied to *execlp()*. You will also need to use functions in the *malloc* package to allocate space for the argument and environment vectors. Finally, note that an easy way of checking whether a file exists in a particular directory and is executable is simply to try execing the file.

27-3. What output would we see if we make the following script executable and *exec()* it?

```
#!/bin/cat -n
Hello world
```

27-4. What is the effect of the following code? In what circumstances might it be useful?

```
childPid = fork();
if (childPid == -1)
    errExit("fork1");
if (childPid == 0) {    /* Child */
    switch (fork()) {
    case -1: errExit("fork2");

    case 0:             /* Grandchild */
        /* ----- Do real work here ----- */
        exit(EXIT_SUCCESS);             /* After doing real work */

    default:
        exit(EXIT_SUCCESS);             /* Make grandchild an orphan */
    }
}

/* Parent falls through to here */

if (waitpid(childPid, &status, 0) == -1)
    errExit("waitpid");

/* Parent carries on to do other things */
```

27-5. When we run the following program, we find it produces no output. Why is this?

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Hello world");
    execlp("sleep", "sleep", "0", (char *) NULL);
}
```

27-6. Suppose that a parent process has established a handler for SIGCHLD and also blocked this signal. Subsequently, one of its children exits, and the parent then does a *wait()* to collect the child's status. What happens when the parent unblocks SIGCHLD? Write a program to verify your answer. What is the relevance of the result for a program calling the *system()* function?