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.