Chapter 5. Process Management

As mentioned in Chapter 1, processes are the most fundamental abstraction in a Unix system, after files. As object code in execution—active, alive, running programs—processes are more than just assembly language; they consist of data, resources, state, and a virtualized computer.

In this chapter, we will look at the fundamentals of the process, from creation to termination. The basics have remained relatively unchanged since the earliest days of Unix. It is here, in the subject of process management, that the longevity and forward thinking of Unix's original design shines brightest. Unix took an interesting path, one seldom traveled, and separated the act of creating a new process from the act of loading a new binary image. Although the two tasks are performed in tandem most of the time, the division has allowed a great deal of freedom for experimentation and evolution for each of the tasks. This road less traveled has survived to this day, and while most operating systems offer a single system call to start up a new program, Unix requires two: a fork and an exec. But before we cover those system calls, let's look more closely at the process itself.

5.1. The Process ID

Each process is represented by a unique identifier, the *process ID* (frequently shortened to *pid*). The pid is guaranteed to be unique at any *single point in time*. That is, while at time `t`0 there can be only one process with the pid 770 (if any process at all exists with such a value), there is no guarantee that at time `t`1 a different process won't exist with pid 770. Essentially, however, most code presumes that the kernel does not readily reissue process identifiers—an assumption that, as you will see shortly, is fairly safe.

The *idle process*—the process that the kernel "runs" when there are no other runnable processes—has the pid 0. The first process that the kernel executes after booting the system, called the *init process*, has the pid 1. Normally, the init process on Linux is the *init* program. We use the term "init" to refer to both the initial process that the kernel runs, and the specific program used for that purpose.

Unless the user explicitly tells the kernel what process to run (through the *init* kernel command-line parameter), the kernel has to identify a suitable init process on its own—a rare example where the kernel dictates policy. The Linux kernel tries four executables, in the following order:

1. */sbin/init*: The preferred and most likely location for the init process.
2. */etc/init*: Another likely location for the init process.
3. */bin/init*: A possible location for the init process.
4. */bin/sh*: The location of the Bourne shell, which the kernel tries to run if it fails to find an init process.

The first of these processes that exists is executed as the init process. If all four processes fail to execute, the Linux kernel halts the system with a panic.

After the handoff from the kernel, the init process handles the remainder of the boot process. Typically, this includes initializing the system, starting various services, and launching a login program.

5.1.1. Process ID Allocation

By default, the kernel imposes a maximum process ID value of 32768. This is for compatibility with older Unix systems, which used smaller 16-bit types for process IDs. System administrators can set the value higher via */proc/sys/kernel/pid_max*, trading a larger pid space for reduced compatibility.

The kernel allocates process IDs to processes in a strictly linear fashion. If pid 17 is the highest number currently allocated, pid 18 will be allocated next, even if the process last assigned pid 17 is no longer running when the new process starts. The kernel does not reuse process ID values until it wraps around from the top—that is, earlier values will not be reused until the value in */proc/sys/kernel/pid_max* is allocated. Therefore, while Linux makes no guarantee of the uniqueness of process IDs over a long period, its allocation behavior does provide at least short-term comfort in the stability and uniqueness of pid values.

5.1.2. The Process Hierarchy

The process that spawns a new process is known as the *parent*; the new process is known as the *child*. Every process is spawned from another process (except, of course, the init process). Therefore, every child has a parent. This relationship is recorded in each process' *parent process ID* (ppid), which is the pid of the child's parent.

Each process is owned by a *user* and a *group*. This ownership is used to control access rights to resources. To the kernel, users and groups are mere integer values. Through the files */etc/passwd* and */etc/group*, these integers are mapped to the human-readable names with which Unix users are familiar, such as the user *root* or the group *wheel* (generally speaking, the Linux kernel has no interest in human-readable strings, and prefers to identify objects with integers). Each child process inherits its parent's user and group ownership.

Each process is also part of a *process group*, which simply expresses its relationship to other processes, and must not be confused with the aforementioned user/group concept. Children normally belong to the same process groups as their parents. In addition, when a shell starts up a pipeline (e.g., when a user enters *ls | less*), all the commands in the pipeline go into the same process group. The notion of a process group makes it easy to send signals to or get information on an entire pipeline, as well as all children of the processes in the pipeline. From the perspective of a user, a process group is closely related to a *job*.

5.1.3. pid_t

Programmatically, the process ID is represented by the `pid_t` type, which is defined in the header file *<sys/types.h>*. The exact backing C type is architecture-specific, and not defined by any C standard. On Linux, however, `pid_t` is generally a typedef to the C `int` type.

5.1.4. Obtaining the Process ID and Parent Process ID

The `getpid( )` system call returns the process ID of the invoking process:

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

pid_t getpid (void);
```

The `getppid( )` system call returns the process ID of the invoking process' parent:

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

pid_t getppid (void);
```

Neither call will return an error. Consequently, usage is trivial:

```
printf ("My pid=%d\n", getpid ( ));
printf ("Parent's pid=%d\n", getppid ( ));
```

How do we know that a `pid_t` is a signed integer? Good question! The answer, simply, is that we do not know. Even though we can safely assume that `pid_t` is an `int` on Linux, such a guess still defeats the intention of the abstract type, and hurts portability. Unfortunately, as with all typedefs in C, there is no easy way to print `pid_t` values—this is part of the abstraction, and technically we need a `pid_to_int( )` function, which we lack. Treating these values as integers, however, at least for the purposes of `printf( )`, is common.

5.2. Running a New Process

In Unix, the act of loading into memory and executing a program image is separate from the act of creating a new process. One system call (actually, one call from a family of calls) loads a binary program into memory, replacing the previous contents of the address space, and begins execution of the new program. This is called *executing* a new program, and the functionality is provided by the *exec* family of calls.

A different system call is used to create a new process, which initially is a near duplicate of its parent process. Often, the new process immediately executes a new program. The act of creating a new process is called *forking*, and this functionality is provided by the `fork( )` system call. Two acts—first a fork, to create a new process, and then an exec, to load a new image into that process—are thus required to execute a new program image in a new process. We will cover the exec calls first, then `fork( )`.

5.2.1. The Exec Family of Calls

There is no single exec function; instead, there is a family of exec functions built on a single system call. Let's first look at the simplest of these calls, `execl( )`:

```
#include <unistd.h>

int execl (const char *path,
           const char *arg,
           ...);
```

A call to `execl( )` replaces the current process image with a new one by loading into memory the program pointed at by `path`. The parameter `arg` is the first argument to this program. The ellipsis signifies a variable number of arguments—the `execl( )` function is *variadic*, which means that additional arguments may optionally follow, one by one. The list of arguments must be `NULL`-terminated.

For example, the following code replaces the currently executing program with */bin/vi*:

```
int ret;

ret = execl ("/bin/vi", "vi", NULL);
if (ret == -1)
        perror ("execl");
```

Note that we follow the Unix convention and pass "vi" as the program's first argument. The shell puts the last component of the path, the "vi," into the first argument when it forks/execs processes, so a program can examine its first argument, `argv[0]`, to discover the name of its binary image. In many cases, several system utilities that appear as different names to the user are in fact a single program with hard links for their multiple names. The program uses the first argument to determine its behavior.

As another example, if you wanted to edit the file */home/kidd/hooks.txt*, you could execute the following code:

```
int ret;

ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
        perror ("execl");
```

Normally, `execl( )` does not return. A successful invocation ends by jumping to the entry point of the new program, and the just-executed code no longer exists in the process' address space. On error, however, `execl( )` returns `-1`, and sets `errno` to indicate the problem. We will look at the possible `errno` values later in this section.

A successful `execl( )` call changes not only the address space and process image, but certain other attributes of the process:

- Any pending signals are lost.
- Any signals that the process is catching (see Chapter 9) are returned to their default behavior, as the signal handlers no longer exist in the process' address space.
- Any memory locks (see Chapter 8) are dropped.
- Most thread attributes are returned to the default values.
- Most process statistics are reset.
- Anything related to the process' memory, including any mapped files, is dropped.
- Anything that exists solely in user space, including features of the C library, such as `atexit( )` behavior, is dropped.

Many properties of the process, however, do *not* change. For example, the pid, parent pid, priority, and owning user and group all remain the same.

Normally, open files are inherited across an exec. This means the newly executed program has full access to all of the files open in the original process, assuming it knows the file descriptor values. However, this is often not the desired behavior. The usual practice is to close files before the exec, although it is also possible to instruct the kernel to do so automatically via `fcntl( )`.

5.2.1.1. The rest of the family

In addition to `execl( )`, there are five other members of the exec family:

```
#include <unistd.h>

int execlp (const char *file,
            const char *arg,
            ...);

int execle (const char *path,
            const char *arg,
            ...,
            char * const envp[]);

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

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

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

The mnemonics are simple. The `l` and `v` delineate whether the arguments are provided via a *l*ist or an array (*v*ector). The `p` denotes that the user's full *p*ath is searched for the given file. Commands using the `p` variants can specify just a filename, so long as it is located in the user's path. Finally, the `e` notes that a new environment is also supplied for the new process. Curiously, although there is no technical reason for the omission, the exec family contains no member that both searches the path and takes a new environment. This is probably because the `p` variants were implemented for use by shells, and shell-executed processes generally inherit their environments from the shell.

The members of the exec family that accept an array work about the same, except that an array is constructed and passed in instead of a list. The use of an array allows the arguments to be determined at runtime. Like the variadic list of arguments, the array must be `NULL`-terminated.

The following snippet uses `execvp( )` to execute *vi*, as we did previously:

```
const char *args[] = { "vi", "/home/kidd/hooks.txt", NULL };
int ret;

ret = execvp ("vi", args);
if (ret == -1)
        perror ("execvp");
```

Assuming */bin* is in the user's path, this works similarly to the last example.

In Linux, only one member of the exec family is a system call. The rest are wrappers in the C library around the system call. Because variadic system calls would be difficult to implement, at best, and because the concept of the user's path exists solely in user space, the only option for the lone system call is `execve( )`. The system call prototype is identical to the user call.

5.2.1.2. Error values

On success, the exec system calls do not return. On failure, the calls return `-1`, and set `errno` to one of the following values:

`E2BIG`

The total number of bytes in the provided arguments list (`arg`) or environment (`envp`) is too large.

`EACCESS`

The process lacks search permission for a component in `path`; `path` is not a regular file; the target file is not marked executable; or the filesystem on which `path` or `file` resides is mounted `noexec`.

`EFAULT`

A given pointer is invalid.

`EIO`

A low-level I/O error occurred (this is bad).

`EISDIR`

The final component in `path`, or the interpreter, is a directory.

`ELOOP`

The system encountered too many symbolic links in resolving `path`.

`EMFILE`

The invoking process has reached its limit on open files.

`ENFILE`

The system-wide limit on open files has been reached.

`ENOENT`

The target of `path` or `file` does not exist, or a needed shared library does not exist.

`ENOEXEC`

The target of `path` or `file` is an invalid binary, or is intended for a different machine architecture.

`ENOMEM`

There is insufficient kernel memory available to execute a new program.

`ENOTDIR`

A nonfinal component in `path` is not a directory.

`EPERM`

The filesystem on which `path` or `file` resides is mounted `nosuid`, the user is not root, and `path` or `file` has the suid or sgid bit set.

`ETXTBSY`

The target of `path` or `file` is open for writing by another process.

5.2.2. The fork( ) System Call

A new process running the same image as the current one can be created via the `fork( )` system call:

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

pid_t fork (void);
```

A successful call to `fork( )` creates a new process, identical in almost all aspects to the invoking process. Both processes continue to run, returning from `fork( )` as if nothing special had happened.

The new process is called the "child" of the original process, which in turn is called the "parent." In the child, a successful invocation of `fork( )` returns `0`. In the parent, `fork( )` returns the pid of the child. The child and the parent process are identical in nearly every facet, except for a few necessary differences:

- The pid of the child is, of course, newly allocated, and different from that of the parent.
- The child's parent pid is set to the pid of its parent process.
- Resource statistics are reset to zero in the child.
- Any pending signals are cleared, and not inherited by the child (see Chapter 9).
- Any acquired file locks are not inherited by the child.

On error, a child process is not created, `fork( )` returns `-1`, and `errno` is set appropriately. There are two possible `errno` values, with three possible meanings:

`EAGAIN`

The kernel failed to allocate certain resources, such as a new pid, or the `RLIMIT_NPROC` resource limit (rlimit) has been reached (see Chapter 6).

`ENOMEM`

Insufficient kernel memory was available to complete the request.

Use is simple:

```
pid_t pid;

pid = fork (  );
if (pid > 0)
        printf ("I am the parent of pid=%d!\n", pid);
else if (!pid)
        printf ("I am the baby!\n");
else if (pid == -1)
        perror ("fork");
```

The most common usage of `fork( )` is to create a new process in which a new binary image is then loaded—think a shell running a new program for the user or a process spawning a helper program. First the process forks a new process, and then the child executes a new binary image. This "fork plus exec" combination is frequent and simple. The following example spawns a new process running the binary */bin/windlass*:

```
pid_t pid;

pid = fork (  );
if (pid == -1)
        perror ("fork");

/* the child ... */
if (!pid) {
        const char *args[] = { "windlass", NULL };
        int ret;
```

```
        ret = execv ("/bin/windlass", args);
        if (ret == -1) {
                perror ("execv");
                exit (EXIT_FAILURE);
        }
}
```

The parent process continues running with no change, other than that it now has a new child. The call to `execv( )` changes the child to running the */bin/windlass* program.

5.2.2.1. Copy-on-write

In early Unix systems, forking was simple, if not naïve. Upon invocation, the kernel created copies of all internal data structures, duplicated the process' page table entries, and then performed a page-by-page copy of the parent's address space into the child's new address space. But this page-by-page copy was, at least from the standpoint of the kernel, time-consuming.

Modern Unix systems behave more optimally. Instead of a wholesale copy of the parent's address space, modern Unix systems such as Linux employ *copy-on-write* (COW) pages.

Copy-on-write is a lazy optimization strategy designed to mitigate the overhead of duplicating resources. The premise is simple: if multiple consumers request read access to their own copies of a resource, duplicate copies of the resource need not be made. Instead, each consumer can be handed a pointer to the same resource. So long as no consumer attempts to modify its "copy" of the resource, the illusion of exclusive access to the resource remains, and the overhead of a copy is avoided. If a consumer does attempt to modify its copy of the resource, at that point, the resource is transparently duplicated, and the copy is given to the modifying consumer. The consumer, never the wiser, can then modify its copy of the resource while the other consumers continue to share the original, unchanged version. Hence the name: the *copy* occurs only *on write*.

The primary benefit is that if a consumer never modifies its copy of the resource, a copy is never needed. The general advantage of lazy algorithms—that they defer expensive actions until the last possible moment—also applies.

In the specific example of virtual memory, copy-on-write is implemented on a per-page basis. Thus, so long as a process does not modify all of its address space, a copy of the entire address space is not required. At the completion of a fork, the parent and child believe that they each have a unique address space, while in fact they are sharing the parent's original pages—which in turn may be shared with other parent or child processes, and so on!

The kernel implementation is simple. The pages are marked as read-only and as copy-on-write in the kernel's page-related data structures. If either process attempts to modify a page, a page fault occurs. The kernel then handles the page fault by transparently making a copy of the page; at this point, the page's copy-on-write attribute is cleared, and it is no longer shared.

Because modern machine architectures provide hardware-level support for copy-on-write in their memory management units (MMUs), the charade is simple and easy to implement.

Copy-on-write has yet a bigger benefit in the case of forking. Because a large percentage of forks are followed by an exec, copying the parent's address space into the child's address space is often a complete waste of time: if the child summarily executes a new binary image, its previous address space is wiped out. Copy-on-write optimizes for this case.

5.2.2.2. vfork( )

Before the arrival of copy-on-write pages, Unix designers were concerned with the wasteful address-space copy during a fork that is immediately followed by an exec. BSD developers therefore unveiled the vfork( ) system call in 3.0BSD:

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

pid_t vfork (void);
```

A successful invocation of vfork( ) has the same behavior as fork( ), except that the child process must immediately issue a successful call to one of the exec functions, or exit by calling _exit( ) (discussed in the next section). The vfork( ) system call avoids the address space and page table copies by suspending the parent process until the child terminates or executes a new binary image. In the interim, the parent and the child share—without copy-on-write semantics—their address space and page table entries. In fact, the only work done during a vfork( ) is the duplication of internal kernel data structures. Consequently, the child must not modify any memory in the address space.

The vfork( ) system call is a relic, and should never have been implemented on Linux, although it should be noted that even with copy-on-write, vfork( ) is faster than fork( ) because the page table entries need not be copied.[1] Nonetheless, the advent of copy-on-write pages weakens any argument for an alternative to fork( ). Indeed, until the 2.2.0 Linux kernel, vfork( ) was simply a wrapper around fork( ). As the requirements for vfork( ) are weaker than the requirements for fork( ), such a vfork( ) implementation is feasible.

[1] * Although not currently part of the 2.6 Linux kernel, a patch implementing copy-on-write shared page table entries has been floated on the Linux Kernel Mailing List (*lkml*). Should it be merged, there would be absolutely no benefit to using vfork( ).

Strictly speaking, no vfork( ) implementation is bug-free: consider the situation if the exec call were to fail! The parent would be suspended indefinitely while the child figured out what to do or until it exited.

5.3. Terminating a Process

POSIX and C89 both define a standard function for terminating the current process:

```
#include <stdlib.h>

void exit (int status);
```

A call to exit( ) performs some basic shutdown steps, and then instructs the kernel to terminate the process. This function has no way of returning an error—in fact, it never returns at all. Therefore, it does not make sense for any instructions to follow the exit( ) call.

The status parameter is used to denote the process' exit status. Other programs—as well as the user at the shell—can check this value. Specifically, status & 0377 is returned to the parent. We will look at retrieving the return value later in this chapter.

`EXIT_SUCCESS` and `EXIT_FAILURE` are defined as portable ways to represent success and failure. On Linux, `0` typically represents success; a nonzero value, such as `1` or `-1`, corresponds to failure.

Consequently, a successful exit is as simple as this one-liner:

```
exit (EXIT_SUCCESS);
```

Before terminating the process, the C library performs the following shutdown steps, in order:

1. Call any functions registered with `atexit( )` or `on_exit( )`, in the reverse order of their registration. (We will discuss these functions later in this chapter.)
2. Flush all open standard I/O streams (see Chapter 3).
3. Remove any temporary files created with the `tmpfile( )` function.

These steps finish all the work the process needs to do in user space, so `exit( )` invokes the system call `_exit( )` to let the kernel handle the rest of the termination process:

```
#include <unistd.h>

void _exit (int status);
```

When a process exits, the kernel cleans up all of the resources that it created on the process' behalf that are no longer in use. This includes, but is not limited to, allocated memory, open files, and System V semaphores. After cleanup, the kernel destroys the process and notifies the parent of its child's demise.

Applications can call `_exit( )` directly, but such a move seldom makes sense: most applications need to do some of the cleanup provided by a full exit, such as flushing the *stdout* stream. Note, however, that `vfork( )` users should call `_exit( )`, and not `exit( )`, after a fork.

> In a brilliant stroke of redundancy, the ISO C99 standard added the `_Exit( )` function, which has identical behavior to `_exit( )`:
>
> ```
> #include <stdlib.h>
>
> void _Exit (int status);
> ```

5.3.1. Other Ways to Terminate

The classic way to end a program is not via an explicit system call, but by simply "falling off the end" of the program. In the case of C, this happens when the `main( )` function returns. The "falling off the end" approach, however, still invokes a system call: the compiler simply inserts an implicit `_exit( )` after its own shutdown code. It is good coding practice to explicitly return an exit status, either via `exit( )`, or by returning a value from `main( )`. The shell uses the exit value for evaluating the success or failure of commands. Note that a successful return is `exit(0)`, or a return from `main( )` of `0`.

A process can also terminate if it is sent a signal whose default action is to terminate the process. Such signals include `SIGTERM` and `SIGKILL` (see Chapter 9).

A final way to end a program's execution is by incurring the wrath of the kernel. The kernel can kill a process for executing an illegal instruction, causing a segmentation violation, running out of memory, and so on.

5.3.2. atexit( )

POSIX 1003.1-2001 defines, and Linux implements, the `atexit( )` library call, used to register functions to be invoked on process termination:

```
#include <stdlib.h>

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

A successful invocation of `atexit( )` registers the given function to run during normal process termination; i.e., when a process is terminated via either `exit( )` or a return from `main( )`. If a process invokes an exec function, the list of registered functions is cleared (as the functions no longer exist in the new process' address space). If a process terminates via a signal, the registered functions are not called.

The given function takes no parameters, and returns no value. A prototype has the form:

```
void my_function (void);
```

Functions are invoked in the reverse order that they are registered. That is, the functions are stored in a stack, and the last in is the first out (LIFO). Registered functions must not call `exit( )`, lest they begin an endless recursion. If a function needs to end the termination process early, it should call `_exit( )`. Such behavior is not recommended, however, as a possibly important function may then not run.

The POSIX standard requires that `atexit( )` support at least `ATEXIT_MAX` registered functions, and that this value has to be at least 32. The exact maximum may be obtained via `sysconf( )` and the value of `_SC_ATEXIT_MAX`:

```
long atexit_max;

atexit_max = sysconf (_SC_ATEXIT_MAX);
printf ("atexit_max=%ld\n", atexit_max);
```

On success, `atexit( )` returns `0`. On error, it returns `-1`.

Here's a simple example:

```
#include <stdio.h>
#include <stdlib.h>

void out (void)
{
        printf ("atexit(  ) succeeded!\n");
}
```

```
int main (void)
{
        if (atexit (out))
                fprintf(stderr, "atexit(  ) failed!\n");

        return 0;
}
```

5.3.3. on_exit( )

SunOS 4 defined its own equivalent to `atexit( )`, and Linux's *glibc* supports it:

```
#include <stdlib.h>

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

This function works the same as `atexit( )`, but the registered function's prototype is different:

```
void my_function (int status, void *arg);
```

The `status` argument is the value passed to `exit( )` or returned from `main( )`. The `arg` argument is the second parameter passed to `on_exit( )`. Care must be taken to ensure that the memory pointed at by `arg` is valid when the function is ultimately invoked.

The latest version of Solaris no longer supports this function. You should use the standards-compliant `atexit( )` instead.

5.3.4. SIGCHLD

When a process terminates, the kernel sends the signal `SIGCHLD` to the parent. By default, this signal is ignored, and no action is taken by the parent. Processes can elect to handle this signal, however, via the `signal( )` or `sigaction( )` system calls. These calls, and the rest of the wonderful world of signals, are covered in Chapter 9.

The `SIGCHLD` signal may be generated and dispatched at any time, as a child's termination is asynchronous with respect to its parent. But often, the parent wants to learn more about its child's termination, or even explicitly wait for the event's occurrence. This is possible with the system calls discussed next.

5.4. Waiting for Terminated Child Processes

Receiving notification via a signal is nice, but many parents want to obtain more information when one of their child processes terminates—for example, the child's return value.

If a child process were to entirely disappear when terminated, as one might expect, no remnants would remain for the parent to investigate. Consequently, the original designers of Unix decided that when a child dies before its parent, the kernel should put the child into a special process state. A process in this state is known as a *zombie*. Only a minimal skeleton of what was once the process—some basic kernel data structures containing potentially useful data—is retained. A process in this state waits for its parent to inquire about its status (a procedure known as *waiting on* the zombie process). Only after the parent obtains the information preserved about the terminated child does the process formally exit and cease to exist even as a zombie.

The Linux kernel provides several interfaces for obtaining information about terminated children. The simplest such interface, defined by POSIX, is `wait( )`:

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

pid_t wait (int *status);
```

A call to `wait( )` returns the pid of a terminated child, or `-1` on error. If no child has terminated, the call blocks until a child terminates. If a child has already terminated, the call returns immediately. Consequently, a call to `wait( )` in response to news of a child's demise—say, upon receipt of a `SIGCHLD`—will always return without blocking.

On error, there are two possible `errno` values:

`ECHILD`

The calling process does not have any children.

`EINTR`

A signal was received while waiting, and the call returned early.

If not `NULL`, the `status` pointer contains additional information about the child. Because POSIX allows implementations to define the bits in `status` as they see fit, the standard provides a family of macros for interpreting the parameter:

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

int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);

int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

Either of the first two macros may return true (a nonzero value), depending on how the process terminated. The first, `WIFEXITED`, returns true if the process terminated normally—that is, if the process called `_exit( )`. In this case, the macro `WEXITSTATUS` provides the low-order eight bits that were passed to `_exit( )`.

`WIFSIGNALED` returns true if a signal caused the process' termination (see Chapter 9 for further discussion on signals). In this case, `WTERMSIG` returns the number of the signal that caused the termination, and `WCOREDUMP` returns true if the process dumped core in response to receipt of the signal. WCOREDUMP is not defined by POSIX, although many Unix systems, Linux included, support it.

WIFSTOPPED and WIFCONTINUED return true if the process was stopped or continued, respectively, and is currently being traced via the ptrace( ) system call. These conditions are generally applicable only when implementing a debugger, although when used with waitpid( ) (see the following subsection), they are used to implement job control, too. Normally, wait( ) is used only to communicate information about a process' termination. If WIFSTOPPED is true, WSTOPSIG provides the number of the signal that stopped the process. WIFCONTINUED is not defined by POSIX, although future standards define it for waitpid( ). As of the 2.6.10 Linux kernel, Linux provides this macro for wait( ), too.

Let's look at an example program that uses wait( ) to figure out what happened to its child:

Code View: Scroll / Show All

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

int main (void)
{
        int status;
        pid_t pid;

        if (!fork (  ))
                return 1;

        pid = wait (&status);
        if (pid == -1)
                perror ("wait");

        printf ("pid=%d\n", pid);

        if (WIFEXITED (status))
                printf ("Normal termination with exit status=%d\n",
                        WEXITSTATUS (status));

        if (WIFSIGNALED (status))
                printf ("Killed by signal=%d%s\n",
                        WTERMSIG (status),
                        WCOREDUMP (status) ? " (dumped core)" : "");

        if (WIFSTOPPED (status))
                printf ("Stopped by signal=%d\n",
                        WSTOPSIG (status));

        if (WIFCONTINUED (status))
                printf ("Continued\n");

       return 0;
}
```

This program forks a child, which immediately exits. The parent process then executes the wait( ) system call to determine the status of its child. The process prints the child's pid, and how it died. Because in this case the child terminated by returning from main( ), we know that we will see output similar to the following:

```
$ ./wait
pid=8529
Normal termination with exit status=1
```

If, instead of having the child return, we have it call `abort( )`, [1] which sends itself the `SIGABRT` signal, we will instead see something resembling the following:

[1] * Defined in the header `<stdlib.h>`.

```
$ ./wait
pid=8678
Killed by signal=6
```

5.4.1. Waiting for a Specific Process

Observing the behavior of child processes is important. Often, however, a process has multiple children, and does not wish to wait for all of them, but rather for a specific child process. One solution would be to make multiple invocations of `wait( )`, each time noting the return value. This is cumbersome, though—what if you later wanted to check the status of a different terminated process? The parent would have to save all of the `wait( )` output, in case it needed it later.

If you know the pid of the process you want to wait for, you can use the `waitpid( )` system call:

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

pid_t waitpid (pid_t pid, int *status, int options);
```

The `waitpid( )` call is a more powerful version of `wait( )`. Its additional parameters allow for fine-tuning.

The `pid` parameter specifies exactly which process or processes to wait for. Its values fall into four camps:

`< -1`

Wait for any child process whose process group ID is equal to the absolute value of this value. For example, passing `-500` waits for any process in process group 500.

`-1`

Wait for any child process. This is the same behavior as `wait( )`.

`0`

Wait for any child process that belongs to the same process group as the calling process.

`> 0`

Wait for any child process whose pid is exactly the value provided. For example, passing `500` waits for the child process with pid 500.

The `status` parameter works identically to the sole parameter to `wait( )`, and can be operated on using the macros discussed previously.

The `options` parameter is a binary OR of zero or more of the following options:

`WNOHANG`

Do not block, but return immediately if no matching child process has already terminated (or stopped or continued).

`WUNTRACED`

If set, `WIFSTOPPED` is set, even if the calling process is not tracing the child process. This flag allows for the implementation of more general job control, as in a shell.

`WCONTINUED`

If set, `WIFCONTINUED` is set even if the calling process is not tracing the child process. As with `WUNTRACED`, this flag is useful for implementing a shell.

On success, `waitpid( )` returns the pid of the process whose state has changed. If `WNOHANG` is specified, and the specified child or children have not yet changed state, `waitpid( )` returns `0`. On error, the call returns `-1`, and `errno` is set to one of three values:

`ECHILD`

The process or processes specified by the `pid` argument do not exist, or are not children of the calling process.

`EINTR`

The `WNOHANG` option was not specified, and a signal was received while waiting.

`EINVAL`

The `options` argument is invalid.

As an example, assume your program wants to grab the return value of the specific child with pid 1742 but return immediately if the child has not yet terminated. You might code up something similar to the following:

```
int status;
pid_t pid;

pid = waitpid (1742, &status, WNOHANG);
if (pid == -1)
        perror ("waitpid");
else {
        printf ("pid=%d\n", pid);

        if (WIFEXITED (status))
                printf ("Normal termination with exit status=%d\n",
                        WEXITSTATUS (status));

        if (WIFSIGNALED (status))
                printf ("Killed by signal=%d%s\n",
                        WTERMSIG (status),
                        WCOREDUMP (status) ? " (dumped core)" : "");
}
```

As a final example, note that the following usage of `wait( )`:

```
wait (&status);
```

is identical to the following usage of `waitpid( )`:

```
waitpid (-1, &status, 0);
```

5.4.2. Even More Waiting Versatility

For applications that require even greater versatility in their waiting-for-children functionality, the XSI extension to POSIX defines, and Linux provides, `waitid( )`:

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

int waitid (idtype_t idtype,
            id_t id,
            siginfo_t *infop,
            int options);
```

As with `wait( )` and `waitpid( )`, `waitid( )` is used to wait for and obtain information about the status change (termination, stopping, continuing) of a child process. It provides even more options, but it offers them with the tradeoff of greater complexity.

Like `waitpid( )`, `waitid( )` allows the developer to specify what to wait for. However, `waitid( )` accomplishes this task with not one, but two parameters. The `idtype` and `id` arguments specify which children to wait for, accomplishing the same goal as the sole `pid` argument in `waitpid( )`. `idtype` may be one of the following values:

`P_PID`

Wait for a child whose pid matches `id`.

`P_GID`

Wait for a child whose process group ID matches `id`.

`P_ALL`

Wait for any child; `id` is ignored.

The `id` argument is the rarely seen `id_t` type, which is a type representing a generic identification number. It is employed in case future implementations add a new `idtype` value, and supposedly offers greater insurance that the predefined type will be able to hold the newly created identifier. The type is guaranteed to be sufficiently large to hold any `pid_t`. On Linux, developers may use it as if it were a `pid_t`—for example, by directly passing `pid_t` values, or numeric constants. Pedantic programmers, however, are free to typecast.

The `options` parameter is a binary OR of one or more of the following values:

`WEXITED`

The call will wait for children (as determined by `id` and `idtype`) that have terminated.

`WSTOPPED`

The call will wait for children that have stopped execution in response to receipt of a signal.

WCONTINUED

The call will wait for children that have continued execution in response to receipt of a signal.

WNOHANG

The call will never block, but will return immediately if no matching child process has already terminated (or stopped, or continued).

WNOWAIT

The call will not remove the matching process from the zombie state. The process may be waited upon in the future.

Upon successfully waiting for a child, `waitid( )` fills in the `infop` parameter, which must point to a valid `siginfo_t` type. The exact layout of the `siginfo_t` structure is implementation-specific, [2] but a handful of fields are valid after a call to `waitid( )`. That is, a successful invocation will ensure that the following fields are filled in:

[2] * Indeed, the `siginfo_t` structure is very complicated on Linux. For its definition, see */usr/include/bits/siginfo.h*. We will study this structure in more detail in Chapter 9.

si_pid

The child's pid.

si_uid

The child's uid.

si_code

Set to one of `CLD_EXITED`, `CLD_KILLED`, `CLD_STOPPED`, or `CLD_CONTINUED` in response to the child terminating, dying via signal, stopping via signal, or continuing via signal, respectively.

si_signo

Set to `SIGCHLD`.

`si_status`

If `si_code` is `CLD_EXITED`, this field is the exit code of the child process. Otherwise, this field is the number of the signal delivered to the child that caused the state change.

On success, `waitid( )` returns `0`. On error, `waitid( )` returns `-1`, and `errno` is set to one of the following values:

`ECHLD`

The process or processes delineated by `id` and `idtype` do not exist.

`EINTR`

`WNOHANG` was not set in `options`, and a signal interrupted execution.

`EINVAL`

The `options` argument or the combination of the `id` and `idtype` arguments is invalid.

The `waitid( )` function provides additional, useful semantics not found in `wait( )` and `waitpid( )`. In particular, the information retrievable from the `siginfo_t` structure may prove quite valuable. If such information is not needed, however, it may make more sense to stick to the simpler functions, which are supported on a wider range of systems, and thus are portable to more non-Linux systems.

5.4.3. BSD Wants to Play: wait3( ) and wait4( )

While `waitpid( )` derives from AT&T's System V Release 4, BSD takes its own route, and provides two other functions used to wait for a child to change state:

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

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

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

The `3` and `4` come from the fact that these two functions are three- and four-parameter versions, respectively, of `wait( )`.

The functions work similarly to `waitpid( )`, with the exception of the `rusage` argument. The following `wait3( )` invocation:

```
pid = wait3 (status, options, NULL);
```

is equivalent to the following `waitpid( )` call:

```
pid = waitpid (-1, status, options);
```

And the following `wait4( )` invocation:

```
pid = wait4 (pid, status, options, NULL);
```

is equivalent to this `waitpid( )` call:

```
pid = waitpid (pid, status, options);
```

That is, `wait3( )` waits for any child to change state, and `wait4( )` waits for the specific child identified by the `pid` parameter to change state. The `options` argument behaves the same as with `waitpid( )`.

As mentioned earlier, the big difference between these calls and `waitpid( )` is the `rusage` parameter. If it is non-`NULL`, the function fills out the pointer at `rusage` with information about the child. This structure provides information about the child's resource usage:

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

struct rusage {
        struct timeval ru_utime; /* user time consumed */
        struct timeval ru_stime; /* system time consumed */
        long ru_maxrss;   /* maximum resident set size */
        long ru_ixrss;    /* shared memory size */
        long ru_idrss;    /* unshared data size */
        long ru_isrss;    /* unshared stack size */
        long ru_minflt;   /* page reclaims */
        long ru_majflt;   /* page faults */
        long ru_nswap;    /* swap operations */
        long ru_inblock;  /* block input operations */
        long ru_oublock;  /* block output operations */
        long ru_msgsnd;   /* messages sent */
        long ru_msgrcv;   /* messages received */
        long ru_nsignals; /* signals received */
        long ru_nvcsw;    /* voluntary context switches */
        long ru_nivcsw;   /* involuntary context switches */
};
```

I will address resource usage further in the next chapter.

On success, these functions return the pid of the process that changed state. On failure, they return `-1`, and set `errno` to one of the same error values returned by `waitpid( )`.

Because wait3( ) and wait4( ) are not POSIX-defined,[3] it is advisable to use them only when resource-usage information is critical. Despite the lack of POSIX standardization, however, nearly every Unix system supports these two calls.

[3] * wait3( ) was included in the original Single UNIX Specification, but it has since been removed.

5.4.4. Launching and Waiting for a New Process

Both ANSI C and POSIX define an interface that couples spawning a new process and waiting for its termination—think of it as synchronous process creation. If a process is spawning a child only to immediately wait for its termination, it makes sense to use this interface:

```
#define _XOPEN_SOURCE    /* if we want WEXITSTATUS, etc. */
#include <stdlib.h>

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

The system( ) function is so named because the synchronous process invocation is called *shelling out to the system*. It is common to use system( ) to run a simple utility or shell script, often with the explicit goal of simply obtaining its return value.

A call to system( ) invokes the command provided by the command parameter, including any additional arguments. The command parameter is suffixed to the arguments */bin/sh -c*. In this sense, the parameter is passed wholesale to the shell.

On success, the return value is the return status of the command as provided by wait( ). Consequently, the exit code of the executed command is obtained via WEXITSTATUS. If invoking */bin/sh* itself failed, the value given by WEXITSTATUS is the same as that returned by exit(127). Because it is also possible for the invoked command to return 127, there is no surefire method to check whether the shell itself returned that error. On error, the call returns -1.

If command is NULL, system( ) returns a nonzero value if the shell */bin/sh* is available, and 0 otherwise.

During execution of the command, SIGCHLD is blocked, and SIGINT and SIGQUIT are ignored. Ignoring SIGINT and SIGQUIT has several implications, particularly if system( ) is invoked inside a loop. If calling system( ) from within a loop, you should ensure that the program properly checks the exit status of the child. For example:

```
do {
        int ret;

        ret = system ("pidof rudderd");
        if (WIFSIGNALED (ret) &&
            (WTERMSIG (ret) == SIGINT ||
             WTERMSIG (ret) == SIGQUIT))
                break; /* or otherwise handle */
} while (1);
```

Implementing system( ) using fork( ), a function from the exec family, and waitpid( ) is a useful exercise. You should attempt this yourself, as it ties together many of the concepts of this chapter. In the spirit of completeness, however, here is a sample implementation:

Code View: Scroll / Show All

```
/*
 * my_system - synchronously spawns and waits for the command
 * "/bin/sh -c <cmd>".
 *
 * Returns -1 on error of any sort, or the exit code from the
 * launched process. Does not block or ignore any signals.
 */
int my_system (const char *cmd)
{
        int status;
        pid_t pid;

        pid = fork (  );
        if (pid == -1)
                return -1;
        else if (pid == 0) {
                const char *argv[4];

                argv[0] = "sh";
                argv[1] = "-c";
                argv[2] = cmd;
                argv[3] = NULL;
                execv ("/bin/sh", argv);

                exit (-1);
        }

        if (waitpid (pid, &status, 0) == -1)
                return -1;
        else if (WIFEXITED (status))
                return WEXITSTATUS (status);

        return -1;
}
```

Note that this example does not block or disable any signals, unlike the official `system( )`. This behavior may be better or worse, depending on your program's situation, but leaving at least `SIGINT` unblocked is often smart because it allows the invoked command to be interrupted in the way a user normally expects. A better implementation could add additional pointers as parameters that, when non-`NULL`, signify errors currently differentiable from each other. For example, one might add `fork_failed` and `shell_failed`.

5.4.5. Zombies

As discussed earlier, a process that has terminated, but that has not yet been waited upon by its parent is called a "zombie." Zombie processes continue to consume system resources, although only a small percentage—enough to maintain a mere skeleton of what they once were. These resources remain so that parent processes that want to check up on the status of their children can obtain information relating to the life and termination of those processes. Once the parent does so, the kernel cleans up the process for good and the zombie ceases to exist.

However, anyone who has used Unix for a good while is sure to have seen zombie processes sitting around. These processes, often called *ghosts*, have irresponsible parents. If your application forks a child process, it is your application's responsibility (unless it is short-lived, as you will see shortly) to wait on the child, even if it will merely discard the information gleaned. Otherwise, all of your process' children will become ghosts and live on, crowding the system's process listing, and generating disgust at your application's sloppy implementation.

What happens, however, if the parent process dies before the child, or if it dies before it has a chance to wait on its zombie children? Whenever a process terminates, the Linux kernel walks a list of its children, and *reparents* all of them to the init process (the process with a pid value of 1). This guarantees that no process is ever without an immediate parent. The init process, in turn, periodically waits on all of its children, ensuring that none remain zombies for too long—no ghosts! Thus, if a parent dies before its children or does not wait on its children before exiting, the child processes are eventually reparented to init and waited upon, allowing them to fully exit. Although doing so is still considered good practice, this safeguard means that short-lived processes need not worry excessively about waiting on all of their children.

5.5. Users and Groups

As mentioned earlier in this chapter, and discussed in Chapter 1, processes are associated with users and groups. The user and group identifiers are numeric values represented by the C types `uid_t` and `gid_t`, respectively. The mapping between numeric values and human-readable names—as in the root user having the uid 0—is performed in user space using the files */etc/passwd* and */etc/group*. The kernel deals only with the numeric values.

In a Linux system, a process' user and group IDs dictate the operations that the process may undertake. Processes must therefore run under the appropriate users and groups. Many processes run as the root user. However, best practices in software development encourage the doctrine of *least-privileged* rights, meaning that a process should execute with the minimum level of rights possible. This requirement is dynamic: if a process requires root privileges to perform an operation early in its life, but does not require these extensive privileges thereafter, it should drop root privileges as soon as possible. To this end, many processes—particularly those that need root privileges to carry out certain operations—often manipulate their user or group IDs.

Before we can look at how this is accomplished, we need to cover the complexities of user and group IDs.

5.5.1. Real, Effective, and Saved User and Group IDs

> The following discussion focuses on user IDs, but the situation is identical for group IDs.

There are, in fact, not one, but four user IDs associated with a process: the real, effective, saved, and filesystem user IDs. The *real user ID* is the uid of the user who originally ran the process. It is set to the real user ID of the process' parent, and does not change during an exec call. Normally, the login process sets the real user ID of the user's login shell to that of the user, and all of the user's processes continue to carry this user ID. The superuser (root) may change the real user ID to any value, but no other user can change this value.

The *effective user ID* is the user ID that the process is currently wielding. Permission verifications normally check against this value. Initially, this ID is equal to the real user ID, because when a process forks, the effective user ID of the parent is inherited by the child. Furthermore, when the process issues an exec call, the effective user is usually unchanged. But, it is during the exec call that the key difference between real and effective IDs emerges: by executing a *setuid* (*suid*) binary, the process can change its effective user ID. To be exact,