CHAPTER 10

The Process Model

One of Unix's hallmarks is its process model. It is the key to understanding access rights, the relationships among open files, signals, job control, and most other low-level topics in this book. Linux adopted most of Unix's process model and added new ideas of its own to allow a truly lightweight threads implementation.

10.1 Defining a Process

What exactly is a process? In the original Unix implementations, a process was any executing program. For each program, the kernel kept track of

- The current location of execution (such as waiting for a system call to return from the kernel), often called the program's **context**
- Which files the program had access to
- The program's **credentials** (which user and group owned the process, for example)
- The program's current directory
- Which memory space the program had access to and how it was laid out

A process was also the basic scheduling unit for the operating system. Only processes were allowed to run on the CPU.

10.1.1 Complicating Things with Threads

Although the definition of a process may seem obvious, the concept of threads makes all of this less clear-cut. A thread allows a single program to run in multiple places at the same time. All the threads created (or spun off) by a single program share most of the characteristics that differentiate processes from each other. For example, multiple threads that originate from the same program share information on open files, credentials, current directory, and memory image. As soon as one of the threads modifies a global variable, all the threads see the new value rather than the old one.

Many Unix implementations (including AT&T's canonical System V release) were redesigned to make threads the fundamental scheduling unit for the kernel, and a process became a collection of threads that shared resources. As so many resources were shared among threads, the kernel could switch between threads in the same process more quickly than it could perform a full context switch between processes. This resulted in most Unix kernels having a two-tiered process model that differentiates between threads and processes.

10.1.2 The Linux Approach

Linux took another route, however. Linux context switches had always been extremely fast (on the same order of magnitude as the new "thread switches" introduced in the two-tiered approach), suggesting to the kernel developers that rather than change the scheduling approach Linux uses, they should allow processes to share resources more liberally.

Under Linux, a process is defined solely as a scheduling entity and the only thing unique to a process is its current execution context. It does not imply anything about shared resources, because a process creating a new child process has full control over which resources the two processes share (see the `clone()` system call described on page 153 for details on this). This model allows the traditional Unix process management approach to be retained while allowing a traditional thread interface to be built outside the kernel.

Luckily, the differences between the Linux process model and the two-tiered approach surface only rarely. In this book, we use the term **process**

to refer to a set of (normally one) scheduling entities which share fundamental resources, and a **thread** is each of those individual scheduling entities. When a process consists of a single thread, we often use the terms interchangeably. To keep things simple, most of this chapter ignores threads completely. Toward the end, we discuss the `clone()` system call, which is used to create threads (and can also create normal processes).

10.2 Process Attributes

10.2.1 The pid and Parentage

Two of the most basic attributes are its **process ID**, or **pid**, and the pid of its parent process. A pid is a positive integer that uniquely identifies a running process and is stored in a variable of type `pid_t`. When a new process is created, the original process is known as the **parent** of the new process and is notified when the new child process ends.

When a process dies, its exit code is stored until the parent process requests it. The exit status is kept in the kernel's process table, which forces the kernel to keep the process's entry active until it can safely discard its exit status. Processes that have exited and are kept around only to preserve their exit status are known as **zombies**. Once the zombie's exit status has been collected, the zombie is removed from the system's process table.

If a process's parent exits (making the child process an **orphan process**), that process becomes a child of the **init process**. The init process is the first process started when a machine is booted and is assigned a pid of 1. One of the primary jobs of the init process is to collect the exit statuses of processes whose parents have died, allowing the kernel to remove the child processes from the process table. Processes can find their pid and their parent's pid through the `getpid()` and `getppid()` functions.

`pid_t getpid(void)`
: Returns the pid of the current process.

`pid_t getppid(void)`
: Returns the parent process's pid.

10.2.2 Credentials

Linux uses the traditional Unix security mechanisms of users and groups. User IDs (**uids**) and group IDs (**gids**) are integers[1] that are mapped to symbolic user names and group names through /etc/passwd and /etc/group, respectively (see Chapter 28 for more information on the user and group databases). However, the kernel knows nothing about the names—it is concerned only with the integer representation. The 0 uid is reserved for the system administrator, normally called **root**. All normal security checks are disabled for processes running as root (that is, with a uid of 0), giving the administrator complete control over the system.

In most cases, a process may be considered to have a single uid and gid associated with it. Those are the IDs that are used for most system security purposes (such as assigning ownerships to newly created files). The system calls that modify a process ownership are discussed later in this chapter.

As Unix developed, restricting processes to a single group turned out to create new difficulties. Users involved in multiple projects had to explicitly switch their gid when they wanted to access files whose access was restricted to members of different groups.

Supplemental groups were introduced in BSD 4.3 to eliminate this problem. Although every process still has its primary gid (which it uses as the gid for newly created files, for example), it also belongs to a set of **supplemental groups**. Security checks that used to ensure that a process belonged to a particular group (and only that group) now allow access as long as the group is one of the supplemental groups to which the process belongs. The `sysconf()` macro `_SC_NGROUPS_MAX` specifies how many supplemental groups a process may belong to. (See Chapter 6, page 54, for details on `sysconf()`.) Under Linux 2.4 and earlier, `_SC_NGROUPS_MAX` is 32; under Linux 2.6 and later, `_SC_NGROUPS_MAX` is 65536. Do not use a static array to store supplemental groups; instead, dynamically allocate memory, taking into account the return value of `sysconf(_SC_NGROUPS_MAX)`. Older code may use the `NGROUPS_MAX` macro to determine how many supplemental

1. uids and gids are normally positive integers, but negative ones work. Using -1 as an ID is problematic, however, as many of the uid- and gid-related system calls use -1 as an indication not to modify a value (see `setregid()` on page 114 for an example of this).

groups are supported by a system; using this macro does not function correctly when the code is compiled in one environment and used in another.

Setting the list of groups for a process is done through the `setgroups()` system call and may be done only by processes running with root permissions.

```
int setgroups(size_t num, const gid_t * list);
```

The `list` parameter points to an array of `num` gids. The process's supplemental group list is set to the gids listed in the `list` array.

The `getgroups()` function lets a process get a list of the supplemental groups to which it belongs.

```
int getgroups(size_t num, gid_t * list);
```

The `list` must point to an array of `gid_t`, which is filled in with the process's supplemental group list, and `num` specifies how many `gid_t`s `list` can hold. The `getgroups()` system call returns -1 on error (normally, if `list` is not large enough to hold the supplemental group list) or the number of supplemental groups. As a special case, if `num` is 0, `getgroups()` returns the number of supplemental groups for the process.

Here is an example of how to use `getgroups()`:

```
gid_t *groupList;
int numGroups;

numGroups = getgroups(0, groupList);
if (numGroups) {
    groupList = alloca(numGroups * sizeof(gid_t));
    getgroups(numGroups, groupList);
}
```

A more complete example of `getgroups()` appears in Chapter 28.

Thus, a process has a uid, a primary gid, and a set of supplemental groups associated with it. Luckily, this is as much as most programmers ever have to worry about. There are two classes of programs that need very flexible management of uids and gids, however: setuid/setgid programs and system daemons.

System daemons are programs that are always running on a system and perform some action at the request of an external stimulus. For example, most World Wide Web (http) daemons are always running, waiting for a client to connect to it so that they can process the client's requests. Other daemons, such as the cron daemon (which runs requests periodically), sleep until a time when they need to perform actions. Most daemons need to run with root permissions but perform actions at the request of a user who may be trying to compromise system security through the daemon.

The ftp daemon is a good example of a daemon that needs flexible uid handling. It initially runs as root and then switches its uid to the uid of the user who logs into it (most systems start a new process to handle each ftp request, so this approach works quite well). This leaves the job of validating file accesses to the kernel where it belongs. Under some circumstances, however, the ftp daemon must open a network connection in a way that only root is allowed to do (see Chapter 17 for details on this). The ftp daemon cannot simply switch back to root, because user processes cannot give themselves superuser access (with good reason!), but keeping the root uid rather than switching to the user's uid would require the ftp daemon to check all file-system accesses itself. The solution to this dilemma has been applied symmetrically to both uids and primary gids, so we just talk about uids here.

A process actually has three uids: its **real uid**, **saved uid**, and **effective uid**.[2] The effective uid is used for all security checks and is the only uid of the process that normally has any effect.

The saved and real uids are checked only when a process attempts to change its effective uid. Any process may change its effective uid to the same value as either its saved or real uid. Only processes with an effective uid of 0 (processes running as root) may change their effective uid to an arbitrary value.

Normally, a process's effective, real, and saved uid's are all the same. However, this mechanism solves the ftp daemon's dilemma. When it starts, all its IDs are set to 0, giving it root permissions. When a user logs in, the daemon sets its effective uid to the uid of the user, leaving both the saved and real uids as 0. When the ftp daemon needs to perform an action

2. Linux processes also have a fourth uid and gid used for file system accesses. This is discussed on page 113.

restricted to root, it sets its effective uid to 0, performs the action, and then resets the effective uid to the uid of the user who is logged in.

Although the ftp daemon does not need the saved uid at all, the other class of programs that takes advantage of this mechanism, setuid and setgid binaries, does use it.

The passwd program is a simple example of why setuid and setgid functionality was introduced. The passwd program allows users to change their passwords. User passwords are usually stored in /etc/passwd. Only the root user may write to this file, preventing other users from changing user information. Users should be able to change their own passwords, however, so some way is needed to give the passwd program permission to modify /etc/passwd.

To allow this flexibility, the owner of a program may set special bits in the program's permission bits (see pages 158–162 for more information), which tells the kernel that whenever the program is executed, it should run with the same effective uid (or gid) as the user who owns the program file, regardless of what user runs the program. These programs are called **setuid** (or **setgid**) executables.

Making the passwd program owned by the root user and setting the setuid bit in the program's permission bits lets all users change their passwords. When a user runs passwd, the passwd program is run with an effective user ID of 0, allowing it to modify /etc/passwd and change the user's password. Of course, the passwd program has to be carefully written to prevent unintended side effects. Setuid programs are a popular target for system intruders, as poorly written setuid programs provide a simple way for users to gain unauthorized permissions.

There are many cases in which setuid programs need their special permissions only for short periods of time and would like to switch back to the uid of the actual user for the remainder of the time (like the ftp daemon). As setuid programs have only their effective uid set to the uid of the program's owner, they know the uid of the user who actually ran them (the saved uid), making switching back simple. In addition, they may set their real uid to the setuid value (without affecting the saved uid) and regain those special permissions as needed. In this situation, the effective, saved, and real uid's work together to make system security as simple as possible.

Unfortunately, using this mechanism can be confusing because POSIX and BSD take slightly different approaches, and Linux supports both. The BSD solution is more full-featured than the POSIX method. It is accessed with the `setreuid()` function.

```
int setreuid(uid_t ruid, uid_t euid);
```

The real uid of the process is set to `ruid` and the effective uid of the process is set to `euid`. If either of the parameters is -1, that ID is not affected by this call.

If the effective uid of the process is 0, this call always succeeds. Otherwise, the IDs may be set to either the saved uid or the real uid of the process. Note that this call never changes the saved uid of the current process; to do that, use POSIX's `setuid()` function, which can modify the saved uid.

```
int setuid(uid_t euid);
```

As in `setreuid()`, the effective uid of the process is set to `euid` as long as `euid` is the same as either the saved or real uid of the process, or as long as the process's effective uid at the time of the call is 0.

When `setuid()` is used by a process whose effective uid is 0, all the process's uids are changed to `euid`. Unfortunately, this makes it impossible for `setuid()` to be used in a setuid root program that needs to temporarily assume the permissions of another uid, because after calling `setuid()`, the process cannot recover its root privileges.

Although the ability to switch uids does make it easier to write code that cannot be tricked into compromising system security, it is not a panacea. There are many popular methods for tricking programs into executing arbitrary code [Lehey, 1995]. As long as either the saved or real uid of a process is 0, such attacks can easily set the effective uid of a process to 0. This makes switching uids insufficient to prevent serious vulnerabilities in system programs. However, if a process can give up any access to root privileges by setting its effective, real, and saved IDs to non-0 values, doing so limits the effectiveness of any attack against it.

10.2.3 The filesystem uid

In highly specialized circumstances, a program may want to keep its root permissions for everything but file-system access, for which it would rather use a user's uid. The user-space NFS server originally used by Linux illustrates the problem that can occur when a process assumes a user's uid. Although the NFS server in the past used `setreuid()` to switch uids while accessing the file system, doing so allowed the user whose uid the NFS server was assuming to kill the NFS server. After all, for a moment that user owned the NFS server process. To prevent this type of problem, Linux uses the **filesystem uid** (fsuid) for file-system access checks.

Whenever a process's effective uid is changed, the process's fsuid is set to the process's new effective user ID, making the fsuid transparent to most applications. Those applications that need the extra features provided by the separate fsuid must use the `setfsuid()` call to explicitly set the fsuid.

```
int setfsuid(uid_t uid);
```

The fsuid may be set to any of the current effective, saved, or real uids of the process. In addition, `setfsuid()` succeeds if the current fsuid is being retained or if the process's effective uid is 0.

10.2.4 User and Group ID Summary

Here is a summary of all the system calls that modify the access permissions of a running process. Most of the functions listed here that pertain to user IDs have been discussed in detail earlier in this chapter, but those related to group IDs have not. As those functions mirror similar functions that modify user IDs, their behavior should be clear.

All of these functions return -1 on error and 0 on success, unless otherwise noted. Most of these prototypes are in `<unistd.h>`. Those that are located elsewhere are noted below.

`int setreuid(uid_t ruid, uid_t euid)`

Sets the real uid of the current process to `ruid` and the effective uid of the process to `euid`. If either of the parameters is -1, that uid remains unchanged.

`int setregid(gid_t rgid, gid_t egid)`
: Sets the real gid of the current process to `rgid` and the effective gid of the process to `egid`. If either of the parameters is -1, that gid remains unchanged.

`int setuid(uid_t uid)`
: If used by a normal user, sets the effective uid of the current process to `uid`. If used by a process with an effective uid of 0, it sets the real, effective, and saved uids to `uid`.

`int setgid(gid_t gid)`
: If used by a normal user, sets the effective gid of the current process to `gid`. If used by a process with an effective gid of 0, sets the real, effective, and saved gids to `gid`.

`int seteuid(uid_t uid)`
: Equivalent to `setreuid(-1, uid)`.

`int setegid(gid_t gid)`
: Equivalent to `setregid(-1, gid)`.

`int setfsuid(uid_t fsuid)`
: Sets the fsuid of the current process to `fsuid`. It is prototyped in `<sys/fsuid.h>`. It returns the previous fsuid.

`int setfsgid(gid_t fsgid)`
: Sets the fsgid of the current process to `fsgid`. It is prototyped in `<sys/fsuid.h>`. It returns the previous fsgid.

`int setgroups(size_t num, const gid_t * list)`
: Sets the supplemental groups for the current process to the groups specified in the array `list`, which must contain `num` items. The `sysconf()` macro `_SC_NGROUPS_MAX` tells how many groups may be in the list (likely 32 or 65536, depending on the Linux versions you are currently running). The `setgroups()` function is prototyped in `<grp.h>`.

`uid_t getuid()`
: Returns the real uid of the process.

`uid_t geteuid()`
: Returns the effective uid of the process.

`gid_t getgid()`
: Returns the real gid of the process.

`gid_t getegid()`
: Returns the effective gid of the process.

`size_t getgroups(size_t size, gid_t list[])`
: Returns the current set of supplemental groups for the current process in the array `list`. `size` tells how many `gid_t`s the list can hold. If the list is not big enough to hold all of the groups, -1 is returned and `errno` is set to `EINVAL`. Otherwise, the number of groups in the list is returned. If `size` is 0, the number of groups in the list is returned, but `list` is not affected. The `getgroups()` function is prototyped in `<grp.h>`.

10.3 Process Information

The kernel makes available quite a bit of information about each process, and some information is passed to new programs when they are loaded. All of this information forms the execution environment for a process.

10.3.1 Program Arguments

There are two types of values passed to new programs when they are run: **command-line arguments** and **environment variables**. Various conventions have been set for their usage, but the system itself does not enforce any of these conventions. It is a good idea to stay within the conventions, however, to help your program fit into the Unix world.

Command-line arguments are a set of strings passed to the program. Usually, these are the text typed after the command name in a shell (hence the name), with optional arguments beginning with a dash (`-`) character.

Environment variables are a set of name/value pairs. Each pair is represented as a single string of the form *`NAME=VALUE`*, and the set of these strings

makes up the program's **environment**. For example, the current user's home directory is normally contained in the `HOME` environment variable, so programs Joe runs often have `HOME=/home/joe` in their environment.

Both the command-line arguments and environment are made available to a program at startup. The command-line arguments are passed as parameters to the program's `main()` function, whereas a pointer to the environment is stored in the `environ` global variable, which is defined in `<unistd.h>`.[3]

Here is the complete prototype of `main()` in the Linux, Unix, and ANSI/ISO C world:

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

You may be surprised to see that `main()` returns a value (other than `void`). The value `main()` returns is given to the process's parent after the process has exited. By convention, 0 indicates that the process completed successfully, and non-0 values indicate failure. Only the lower eight bits of a process's exit code are considered significant. The negative values between -1 and -128 are reserved for processes that are terminated abnormally by another process or by the kernel. An exit code of 0 indicates successful completion, and exit codes between 1 and 127 indicate the program exited because of an error.

The first parameter, `argc`, contains the number of command-line arguments passed to the program, whereas `argv` is an array of pointers to the argument strings. The first item in the array is `argv[0]`, which contains the name of the program being invoked (although not necessarily the complete path to the program). The next-to-last item in the array `argv[argc - 1]` points to the final command-line argument, and `argv[argc]` contains `NULL`.

To access the environment directly, use the following global variable:

```
extern char * environ[];
```

This provides `environ` as an array of pointers to each element in the program's environment (remember, each element is a `NAME=VALUE` pair), and

3. Most systems pass the environment as a parameter to `main()`, as well, but this method is not standardized by POSIX. The `environ` variable is the POSIX-approved method.

the final item in the array is NULL. This declaration appears in <unistd.h>, so you do not need to declare it yourself. The most common way of checking for elements in the environment is through getenv, which eliminates the need for directly accessing environ.

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

The sole parameter to getenv() is the name of the environment variable whose value you are interested in. If the variable exists, getenv() returns a pointer to its value. If the variable does not exist in the current environment (the environment pointed to by environ), it returns NULL.

Linux provides two ways of adding strings to the program's environment, setenv() and putenv(). POSIX defines only putenv(), making it the more portable of the two.

```
int putenv(const char * string);
```

The string passed must be of the form *NAME*=*VALUE*. putenv() adds a variable named *NAME* to the current environment and gives it the value of *VALUE*. If the environment already contains a variable named *NAME*, its value is replaced with *VALUE*.

BSD defines setenv(), which Linux also provides, a more flexible and easier-to-use method for adding items to the environment.

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

Here, the name and the new value of the environment variable to manipulate are passed separately, which is usually easier for a program to do. If overwrite is 0, the environment is not modified if it already contains a variable called name. Otherwise, the variable's value is modified, as in putenv().

Here is a short example of both of these functions. Both calls accomplish exactly the same thing, changing the current PATH environment variable for the running program.

```
putenv("PATH=/bin:/usr/bin");
setenv("PATH", "/bin:/usr/bin", 1);
```

Table 10.1 Process Resources Tracked by Linux

Type	Member	Description
`struct timeval`	`ru_utime`	Total time spent executing user mode code. This includes all the time spent running instructions in the application, but not the time the kernel spends fulfilling application requests.
`struct timeval`	`ru_stime`	Total time the kernel spent executing requests from the process. This does not include time spent while the process was blocked inside a system call.
`long`	`ru_minflt`	The number of **minor faults** that this process caused. Minor faults are memory accesses that force the processor into kernel mode but do not result in a disk access. These occur when a process tries to write past the end of its stack, forcing the kernel to allocate more stack space before continuing the process, for example.
`long`	`ru_majflt`	The number of **major faults** that this process caused. Major faults are memory accesses that force the kernel to access the disk before the program can continue to run. One common cause of major faults is a process accessing a part of its executable memory that has not yet been loaded into RAM from the disk or has been swapped out at some point.
`long`	`ru_nswap`	The number of memory pages that have been paged in from disk due to memory accesses from the process.

10.3.2 Resource Usage

The Linux kernel tracks how many resources each process is using. Although only a few resources are tracked, their measurement can be useful for developers, administrators, and users. Table 10.1 lists the process resources usage currently tracked by the Linux kernel, as of version 2.6.7.

A process may examine the resource usage of itself, the accumulated usages of all its children, or the sum of the two. The `getrusage()` system call returns a `struct rusage` (which is defined in `<sys/resource.h>`) that contains the current resources used.

```
int getrusage(int who, struct rusage * usage);
```

The first parameter, `who`, tells which of the three available resource counts should be returned. `RUSAGE_SELF` returns the usage for the current process, `RUSAGE_CHILDREN` returns the total usage for all of the current process's children, and `RUSAGE_BOTH` yields the total resources used by this process and all its children. The second parameter to `getrusage()` is a pointer to a `struct rusage`, which gets filled in with the appropriate resource utilizations. Although `struct rusage` contains quite a few members (the list is derived from BSD), most of those members are not yet used by Linux. Here is the complete definition of `struct rusage`. Table 10.1 describes the members currently used by Linux.

```
#include <sys/resource.h>
struct rusage
{
  struct timeval ru_utime;
  struct timeval ru_stime;
  long int ru_maxrss;
  long int ru_ixrss;
  long int ru_idrss;
  long int ru_isrss;
  long int ru_minflt;
  long int ru_majflt;
  long int ru_nswap;
  long int ru_inblock;
  long int ru_oublock;
  long int ru_msgsnd;
  long int ru_msgrcv;
  long int ru_nsignals;
  long int ru_nvcsw;
  long int ru_nivcsw;
};
```

10.3.3 Establishing Usage Limits

To help prevent runaway processes from destroying a system's performance, Unix keeps track of many of the resources a process can use and allows the system administrator and the users themselves to place limits on the resources a process may consume.

There are two classes of limits available: hard limits and soft limits. Hard limits may be lowered by any process but may be raised only by the super user. Hard limits are usually set to `RLIM_INFINITY` on system startup, which means no limit is enforced. The only exception to this is `RLIMIT_CORE` (the maximum size of a core dump), which Linux initializes to 0 to prevent unexpected core dumps. Many distributions reset this limit on startup, however, as most technical users expect core dumps under some conditions (see page 130 for information on what a core dump is). Soft limits are the limits that the kernel actually enforces. Any process may set the soft limit for a resource to any value less than or equal to that process's hard limit for the resource.

The various limits that may be set are listed in Table 10.2 and are defined in `<sys/resource.h>`. The `getrlimit()` and `setrlimit()` system calls get and set the limit for a single resource.

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

Both of these functions use `struct rlimit`, which is defined as follows:

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

struct rlimit {
  long int rlim_cur;              /* the soft limit */
  long int rlim_max;              /* the hard limit */
};
```

The second member, `rlim_max`, indicates the hard limit for the limit indicated by the `resource` parameter, and `rlim_cur` contains the soft limit. These are the same sets of limits manipulated by the `ulimit` and `limit` commands, one or the other of which is built into most shells.

Table 10.2 Resource Limits

Value	Limit
`RLIMIT_AS`	Maximum amount of memory available to the process. This includes memory used for the stack, global variables, and dynamically allocated memory.
`RLIMIT_CORE`	Maximum size of a core file generated by the kernel (if the core file would be too large, none is created).
`RLIMIT_CPU`	Total CPU time used (in seconds). For more information on this limit, see the description of `SIGXCPU` on page 221
`RLIMIT_DATA`	Maximum size of data memory (in bytes). This doesn't include memory dynamically allocated by the program.
`RLIMIT_FSIZE`	Maximum size for an open file (checked on writes). For more information on this limit, see the description of `SIGXFSZ` on page 221.
`RLIMIT_MEMLOCK`	Maximum amount of memory that may be locked with `mlock()` (`mlock()` is decsribed on page 275).
`RLIMIT_NOFILE`	Maximum number of open files.
`RLIMIT_NPROC`	Maximum number of child processes the process may spawn. This limits only how many children the process may have at one time. It does not limit how many descendants it may have—each child may have up to `RLIMIT_NPROC` children of its own.
`RLIMIT_RSS`	Maximum amount of RAM used at any time (any memory usage exceeding this causes paging). This is known as a process's **resident set size**.
`RLIMIT_STACK`	Maximum size of stack memory (in bytes), including all local variables.

10.4 Process Primitives

Despite the relatively long discussion needed to describe a process, creating and destroying processes in Linux is straightforward.

10.4.1 Having Children

Linux has two system calls that create new processes: `fork()` and `clone()`. As mentioned earlier, `clone()` is used for creating threads and is discussed briefly later in this chapter. For now, we focus on `fork()`, which is the most popular method of process creation.

```
#include <unistd.h>

pid_t fork(void);
```

This system call has the unusual property of not returning once per invocation, but twice: once in the parent and once in the child. Note that we did not say "first in the parent"—writing code that makes any assumptions about the two processes executing in a deterministic order is a very bad idea.

Each return from the `fork()` system call returns a different value. In the parent process, the system call returns the pid of the newly created child process; in the child process, the call returns 0.

The difference in return value is the only difference apparent to the processes. Both have the same memory image, credentials, open files,[4] and signal handlers. Here is a simple example of a program that creates a child:

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

int main(void) {
    pid_t child;

    if (!(child = fork())) {
        printf("in child\n");
        exit(0);
    }
    printf("in parent -- child is %d\n", child);
    return 0;
}
```

4. For details on how the parent's and child's open files relate to each other, see page 197.

10.4.2 Watching Your Children Die

Collecting the exit status of a child is called **waiting** on the process. There are four ways this can be done, although only one of the calls is provided by the kernel. The other three methods are implemented in the standard C library. As the kernel system call takes four arguments, it is called `wait4()`.

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

The first argument, `pid`, is the process whose exit code should be returned. It can take on a number of special values.

`< -1` Waits for any child whose pgid is the same as the absolute value of `pid`.

`= -1` Waits for any child to terminate.

`= 0` Waits for a child in the same process group current process.[5]

`> 0` Waits for process `pid` to exit.

The second parameter is a pointer to an integer that gets set to the exit status of the process that caused the `wait4()` to return (which we hereafter call the *examined* process). The format of the returned status is convoluted, and a set of macros is provided to make sense of it.

Three events cause `wait4()` to return the status of the examined process: The process could have exited, it could have been terminated by a `kill()` (sent a fatal signal), or it could have been stopped for some reason.[6] You can find out which of these occurred through the following macros, each of which takes the returned status from `wait4()` as the sole parameter:

`WIFEXITED(status)`
: Returns true if the process exited normally. A process exits normally when its `main()` function returns or the program calls `exit()`. If `WIFEXITED()` is true, `WEXITSTATUS(status)` returns the process's exit code.

5. Process groups are described on pages 136–138.
6. See Chapter 15 for reasons why this might happen.

`WIFSIGNALED(status)`
: Returns true if the process was terminated due to a signal (this is what happens for processes terminated with `kill()`). If this is the case, `WTERMSIG(status)` returns the signal number that terminated the process.

`WIFSTOPPED(status)`
: If the process has been stopped by a signal, `WIFSTOPPED()` returns true and `WSTOPSIG(status)` returns the signal that stopped the process. `wait4()` returns information on stopped processes only if `WUNTRACED` was specified as an option.

The `options` argument controls how the call behaves. `WNOHANG` causes the call to return immediately. If no processes are ready to report their status, the call returns 0 rather than a valid pid. `WUNTRACED` causes `wait4()` to return if an appropriate child has been stopped. See Chapter 15 for more information on stopped processes. Both of these behaviors may be specified by bitwise OR'ing the two values together.

The final parameter to `wait4()`, a pointer to a `struct rusage`, gets filled in with the resource usage of the examined process and all the examined process's children. See the discussion of `getrusage()` and `RUSAGE_BOTH` on pages 118–119 for more information on what this entails. If this parameter is `NULL`, no status information is returned.

There are three other interfaces to `wait4()`, all of which provide subsets of its functionality. Here is a summary of the alternative interfaces.

`pid_t wait(int *status)`
: The only parameter to `wait()` is a pointer to the location to store the terminated process's return code. This function always blocks until a child has terminated.

`pid_t waitpid(pid_t pid, int *status, int options)`
: The `waitpid()` function is similar to `wait4()`; the only difference is that it does not return resource usage information on the terminated process.

`pid_t wait3(int *status, int options, struct rusage *rusage)`
: This function is also similar to `wait4()`, but it does not allow the caller to specify which child should be checked.

10.4.3 Running New Programs

Although there are six ways to run one program from another, they all do about the same thing—replace the currently running program with another. Note the word *replace*—all traces of the currently running program disappear. If you want to have the original program stick around, you must create a new process with `fork()` and then execute the new program in the child process.

The six functions feature only slight differences in the interface. Only one of these functions, `execve()`, is actually a system call under Linux. The rest of the functions are implemented in user-space libraries and utilize `execve()` to execute the new program. Here are the prototypes of the `exec()` family of functions:

```
int execl(const char * path, const char * arg0, ...);
int execlp(const char * file, const char * arg0, ...);
int execle(const char * path, const char * arg0, ...);
int execv(const char * path, const char ** argv);
int execvp(const char * file, const char ** argv);
int execve(const char * file, const char ** argv, const char ** envp);
```

As mentioned, all of these programs try to replace the current program with a new program. If they succeed, they never return (as the program that called them is no longer running). If they fail, they return -1 and the error code is stored in `errno`, as with any other system call. When a new program is run (or `exec()`ed) it gets passed an array of arguments (`argv`) and an array of environment variables (`envp`). Each element in `envp` is of the form *`VARIABLE=value`*.[7]

The primary difference between the various `exec()` functions is how the command line arguments are passed to the new program. The `execl` family passes each element in `argv` (the command-line arguments) as a separate argument to the function, and `NULL` terminates the entire list. Traditionally, the first element in `argv` is the command used to invoke the new program. For example, the shell command `/bin/cat /etc/passwd /etc/group` normally results in the following exec call:

7. This is the same format the command `env` uses to print the current environment variables settings, and the `envp` argument is of the same type as the `environ` global variable.

```
execl("/bin/cat", "/bin/cat", "/etc/passwd", "/etc/group", NULL);
```

The first argument is the full path to the program being executed and the rest of the arguments get passed to the program as `argv`. The final parameter to `execl()` must be `NULL`—it indicates the end of the parameter list. If you omit the `NULL`, the function call is likely to result in either a segmentation fault or return `EINVAL`. The environment passed to the new program is whatever is pointed to by the `environ` global variable, as mentioned on page 116.

The `execv` functions pass the command-line argument as a C array of strings,[8] which is the same format used to pass `argv` to the new program. The final entry in the `argv` array must be `NULL` to indicate the end of the array, and the first element (`argv[0]`) should contain the name of the program that was invoked. Our `./cat /etc/passwd /etc/group` example would be coded using `execv` like this:

```
char * argv[] = { "./cat", "/etc/passwd", "/etc/group", NULL };
execv("/bin/cat", argv);
```

If you need to pass a specific environment to the new program, `execle()` and `execve()` are available. They are exactly like `execl()` and `execv()` but they take a pointer to the environment as their final argument. The environment is set up just like `argv`.

For example, here is one way to execute `/usr/bin/env` (which prints out the environment it was passed) with a small environment:

```
char * newenv[] = { "PATH=/bin:/usr/bin",
                    "HOME=/home/sweethome", NULL };
execle("/usr/bin/env", "/usr/bin/env", NULL, newenv);
```

Here is the same idea implemented with `execve()`:

```
char * argv[] = { "/usr/bin/env", NULL };
char * newenv[] = { "PATH=/bin:/usr/bin",
                    "HOME=/home/sweethome", NULL };
execve("/usr/bin/env", argv, newenv);
```

8. Technically, a pointer to a `NULL`-terminated array of pointers to `'\0'` terminated arrays of characters. If this does not make sense, see [Kernighan, 1988].

The final two functions, `execlp()` and `execvp()`, differ from the first two by searching the current path (set by the `PATH` environment variable) for the program to execute. The arguments to the program are not modified, however, so `argv[0]` does not contain the full path to the program being run. Here are modified versions of our first example that search for `cat` in the current `PATH`:

```
execlp("cat", "cat", "/etc/passwd", "/etc/group", NULL);

char * argv[] = { "cat", "/etc/passwd", "/etc/group", NULL };
execvp("cat", argv);
```

If `execl()` or `execv()` were used instead, those code fragments would fail unless `cat` was located in the current directory.

If you are trying to run a program with a specific environment while still searching the path, you need to search the path manually and use `execle()` or `execve()`, because none of the available `exec()` functions does quite what you want.

Signal handlers are preserved across the `exec()` functions in a slightly nonobvious way; the mechanism is described on page 205.

10.4.4 Faster Process Creation with `vfork()`

Normally processes that `fork()` immediately `exec()` another program (this is what shells do every time you type a command), making the full semantics of `fork()` more computationally expensive than is necessary. To help optimize this common case, `vfork()` is provided.

```
#include <unistd.h>

pid_t vfork(void);
```

Rather than creating an entirely new execution environment for the new process, `vfork()` creates a new process that shares the memory of the original process. The new process is expected to either `_exit()` or `exec()` another process very quickly, and the behavior is undefined if it modifies any memory, returns from the function the `vfork()` is contained in, or calls any new functions. In addition, the original process is suspended

until the new one either terminates or calls an `exec()` function.[9] Not all systems provide the memory-sharing and parent-suspending semantics of `vfork()`, however, and applications should never rely upon that behavior.

10.4.5 Killing Yourself

Processes terminate themselves by calling either `exit()` or `_exit()`. When a process's `main()` function returns, the standard C library calls `exit()` with the value returned from `main()` as the parameter.

```
void exit(int exitCode)
void _exit(int exitCode)
```

The two forms, `exit()` and `_exit()`, differ in that `exit()` is a function in the C library, while `_exit()` is a system call. The `_exit()` system call terminates the program immediately, and the `exitCode` is stored as the exit code of the process. When `exit()` is used, functions registered by `atexit()` are called before the library calls `_exit(exitCode)`. Among other things, this allows the ANSI/ISO standard I/O library to flush all its buffers.

Registering functions to be run when `exit()` is used is done through the `atexit()` function.

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

The only parameter passed to `atexit()` is a pointer to a function. When `exit()` is invoked, all the functions registered with `atexit()` are called in the opposite order from which they were registered. Note that if `_exit()` is used or the process is terminated due to a signal (see Chapter 12 for details on signals), functions registered via `atexit()` are not called.

9. `vfork()` was motivated by older systems that needed to copy all of the memory used by the original process as part of the `fork()`. Modern operating systems use **copy-on-write**, which copies memory regions only as necessary, as discussed in most operating system texts [Vahalia, 1997] [Bach, 1986]. This facility makes `fork()` almost as fast as `vfork()`, and much easier to use.

10.4.6 Killing Others

Destroying other processes is almost as easy as creating a new one—just kill it:

```
int kill(pid_t pid, int signum);
```

`pid` should be the pid of the process to kill, and `signum` describes how to kill it. There are two choices[10] for how to kill a child. You can use `SIGTERM` to terminate the process gently. This means that the process can ask the kernel to tell it when someone is trying to kill it so that it can terminate gracefully (saving files, for example). The process may also ignore this type of request for it to terminate and, instead, continue running. Using `SIGKILL` for the `signum` parameter kills the process immediately, no questions asked. If `signum` is zero, then `kill()` checks to see if the process calling `kill()` has the proper permissions, and returns zero if so and nonzero if its permissions are insufficient. This provides a way for a process to check the validity of a pid.

The `pid` parameter can take on four types of values under Linux.

`pid > 0` The signal is sent to the process whose pid is `pid`. If no process exists with that pid, `ESRCH` is returned.

`pid < -1` The signal is sent to all the processes in the process group whose pgid is `-pid`. For example, `kill(-5316, SIGKILL)` immediately terminates all the processes in process group 5316. This ability is used by job control shells, as discussed in Chapter 15.

`pid = 0` The signal is sent to all the processes in the current process's process group.

`pid = -1` The signal is sent to all the processes on the system except the init process. This is used during system shutdown.

10. This is a gross oversimplification. `kill()` actually sends a signal, and signals are a complicated topic in their own right. See Chapter 12 for a complete description of what signals are and how to use them.

Processes can normally `kill()` only processes that share the same effective user ID as themselves. There are two exceptions to this rule. First, processes with an effective uid of 0 may `kill()` any process on the system. Second, any process can send a `SIGCONT` signal to any other process in the same session.[11]

10.4.7 Dumping Core

Although we just mentioned that passing `SIGTERM` and `SIGKILL` to `kill()` causes a process to terminate, you can use quite a few different values (Chapter 12 discusses all of them). Some of these, such as `SIGABRT`, cause the program to dump **core** before dying. A program's core dump contains a complete history of the state of the program when it died.[12] Most debuggers, including gdb, can analyze a core file and tell you what the program was doing when it died, as well as let you inspect the defunct process's memory image. Core dumps end up in the process's current working directory in a file called (simply enough) `core`.

When a process has violated some of the system's requirements (such as trying to access memory that it is not allowed to access), the kernel terminates the process by calling an internal version of `kill()` with a parameter that causes a core dump. The kernel may kill a process for several reasons, including arithmetic violations, such as division by zero; the programs running illegal instructions; and the programs trying to access inaccessible regions of memory. This last case causes a **segmentation fault**, which results in the message `segmentation fault (core dumped)`. If you do any reasonable amount of Linux programming, you are sure to tire of this message!

If a process's resource limit for core files is 0 (see page 120 for details on the core resource limit), no core file is generated.

11. This is to allow job control shells to restart processes that have changed their effective user ID. See chapter Chapter 15 for more information on job control.
12. A once-popular form of computer memory consists of small iron rings arranged in a matrix, with each ring held in place by two wires that are used to sense and set the magnetic polarity of the ring. Each of the rings is called a core, and the whole thing is core memory. So a core dump is a copy of the state of the system's memory (or core) at a given time.

10.5 Simple Children

Although `fork()`, `exec()`, and `wait()` allow programs to make full use of the Linux process model, many applications do not need that much control over their children. There are two library functions that make it easier to use child processes: `system()` and `popen()`.

10.5.1 Running and Waiting with `system()`

Programs regularly want to run another program and wait for it to finish before continuing. The `system()` function allows a program to do this easily.

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

`system()` forks a child process that `exec()`s `/bin/sh`, which then runs `cmd`. The original process waits for the child shell to exit and returns the same status code that `wait()` does.[13] If you do not need the shell to hang around (which is rarely necessary), `cmd` should contain a preceding `"exec"`, which causes the shell to `exec()` `cmd` rather than run `cmd` as a subprocess.

As `cmd` is run through the `/bin/sh` shell, normal shell rules for command expansion apply. Here is an example of `system()` that displays all the C source files in the current directory.

13. In the process, `system()` blocks `SIGCHLD`, which will cause a `SIGCHLD` signal to be delivered to the program immediately before `system()` returns (but after `system()` has called `wait()` on the process it spawned), so programs that use signal handlers need to be careful to handle this possibly spurious signal. The `system()` function also ignores `SIGINT` and `SIGQUIT`, which means that a tight loop calling `system()` repeatedly could be uninterruptible by everything except `SIGSTOP` and `SIGKILL`.

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

int main() {
    int result;

    result = system("exec ls *.c");

    if (!WIFEXITED(result))
        printf("(abnormal exit)\n");

    exit(0);
}
```

The `system()` command should be used very carefully in programs that run with special permissions. As the system shell provides many powerful features and is strongly influenced by environment variables, `system()` provides many potential security weaknesses for intruders to exploit. As long as the application is not a system daemon or a setuid/setgid program, however, `system()` is perfectly safe.

10.5.2 Reading or Writing from a Process

Although `system()` displays the command's output on standard output and allows the child to read from standard input, this is not always ideal. Often, a process wants to read the output from a process or send it text on standard input. `popen()` makes it easy for a process to do this.[14]

```
FILE * popen(const char * cmd, const char * mode);
```

The `cmd` is run through the shell, just as with `system()`. The `mode` should be `"r"` if the parent wants to read the command's output and `"w"` to write to the child's standard input. Note that you cannot do both with `popen()`;

14. While `popen()` makes this easy, it has some hidden behaviors that are not immediately obvious. It creates a child process that might terminate before `pclose()` is called, which would cause the `wait()` functions to return status on the process. When that process ends, it also causes a `SIGCHLD` to be generated, which could confuse a naively written signal handler.

two processes reading from and writing to each other is complex[15] and beyond `popen()`'s abilities.[16]

`popen()` returns a `FILE *` (as defined by the ANSI/ISO standard I/O library), which can be read from and written to just like any other `stdio` stream,[17] or `NULL` if the operation fails. When the parent process is finished, it should use `pclose()` to close the stream and terminate the child process if it is still running. Like `system()`, `pclose()` returns the child's status from `wait4()`.

```
int pclose(FILE * stream);
```

Here is a simple calculator program that uses the **bc** program to do all of the real work. It is important to flush the `popen()`ed stream after writing to it to prevent `stdio` buffering from delaying output (see [Kernighan, 1988] for details on buffering in the ANSI/ISO C `stdio` library functions).

```
 1: /* calc.c */
 2:
 3: /* This is a very simple calculator which uses the external bc
 4:    command to do everything. It opens a pipe to bc, reads a command
 5:    in, passes it to bc, and exits. */
 6: #include <stdio.h>
 7: #include <sys/wait.h>
 8: #include <unistd.h>
 9:
10: int main(void) {
11:     char buf[1024];
12:     FILE * bc;
13:     int result;
14:
15:     /* open a pipe to bc, and exit if we fail */
16:     bc = popen("bc", "w");
17:     if (!bc) {
```

15. This type of processing often results in **deadlocks**, in which process A is waiting for process B to do something, while process B is waiting for process A, resulting in nothing at all getting done.
16. If you find yourself needing to do this, start the child with `fork()` and `exec()` and use `poll()` to read to and write from the child process. A program called **expect** is designed to do this.
17. For information on reading and writing from `stdio` streams, consult [Kernighan, 1988].

```
18:         perror("popen");
19:         return 1;
20:     }
21:
22:     /* prompt for an expression, and read it in */
23:     printf("expr: "); fflush(stdout);
24:     fgets(buf, sizeof(buf), stdin);
25:
26:     /* send the expression to bc for evaluation */
27:     fprintf(bc, "%s\n", buf);
28:     fflush(bc);
29:
30:     /* close the pipe to bc, and wait for it to exit */
31:     result = pclose(bc);
32:
33:     if (!WIFEXITED(result))
34:         printf("(abnormal exit)\n");
35:
36:     return 0;
37: }
```

Like `system()`, `popen()` runs commands through the system shell and should be used very cautiously by programs that run with root credentials.

10.6 Sessions and Process Groups

In Linux, as in other Unix systems, users normally interact with groups of related processes. Although they initially log in to a single terminal and use a single process (their **shell**, which provides a command-line interface), users end up running many processes as a result of actions such as

- Running noninteractive tasks in the background
- Switching among interactive tasks via **job control**, which is discussed more fully in Chapter 15
- Starting multiple processes that work together through pipes

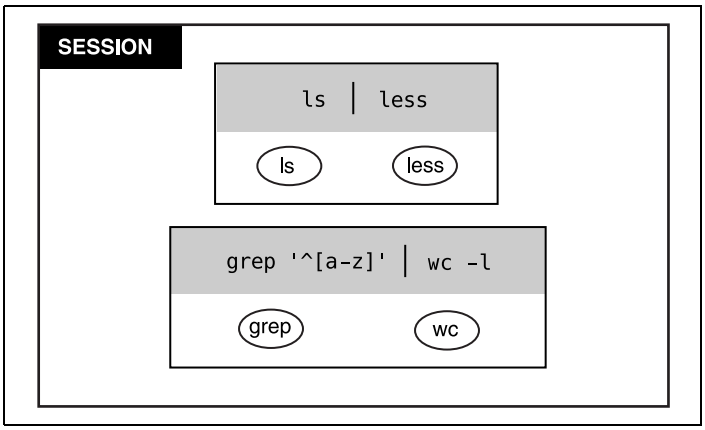

Figure 10.1 Sessions, Process Groups, and Processes

- Running a windowing system, such as the X Window System, which allows multiple terminal windows to be opened

In order to manage all of these processes, the kernel needs to group the processes in ways more complicated than the simple parent-child relationship we have already discussed. These groupings are called **sessions** and **process groups**. Figure 10.1 shows the relationship among sessions, process groups, and processes.

10.6.1 Sessions

When a user logs out of a system, the kernel needs to terminate all the processes the user had running (otherwise, users would leave a slew of old processes sitting around waiting for input that can never arrive). To simplify this task, processes are organized into sets of **sessions**. The session's ID is the same as the pid of the process that created the session through the `setsid()` system call. That process is known as the **session leader** for that session group. All of that process's descendants are then members of that session unless they specifically remove themselves from it. The `setsid()` function does not take any arguments and returns the new session ID.

```
#include <unistd.h>

pid_t setsid(void);
```

10.6.2 Controlling Terminal

Every session is tied to a terminal from which processes in the session get their input and to which they send their output. That terminal may be the machine's local console, a terminal connected over a serial line, or a pseudo terminal that maps to an X window or across a network (see Chapter 16 for information on pseudo terminal devices). The terminal to which a session is related is called the **controlling terminal** (or **controlling tty**) of the session. A terminal can be the controlling terminal for only one session at a time.

Although the controlling terminal for a session can be changed, this is usually done only by processes that manage a user's initial logging in to a system. Information on how to change a session's controlling tty appears in Chapter 16, on pages 338–339.

10.6.3 Process Groups

One of the original design goals of Unix was to construct a set of simple tools that could be used together in complex ways (through mechanisms like pipes). Most Linux users have done something like the following, which is a practical example of this philosophy:

```
ls | grep "^[aA].*\.gz" | more
```

Another popular feature added to Unix fairly early was job control. Job control allows users to suspend the current task (known as the **foreground** task) while they go and do something else on their terminals. When the suspended task is a sequence of processes working together, the system needs to keep track of which processes should be suspended when the user wants to suspend "the" foreground task. Process groups allow the system to keep track of which processes are working together and hence should be managed together via job control.

Processes are added to a process group through `setpgid()`.

```
int setpgid(pid_t pid, pid_t pgid);
```

`pid` is the process that is being placed in a new process group (0 may be used to indicate the current process). `pgid` is the process group ID the

process `pid` should belong to, or 0 if the process should be in a new process group whose pgid is the same as that process's pid. Like sessions, a **process group leader** is the process whose pid is the same as its process group ID (or **pgid**).

The rules for how `setpgid()` may be used are a bit complicated.

1. A process may set the process group of itself or one of its children. It may not change the process group for any other process on the system, even if the process calling `setpgid()` has root privileges.

2. A session leader may not change its process group.

3. A process may not be moved into a process group whose leader is in a different session from itself. In other words, all the processes in a process group must belong to the same session.

`setpgid()` call places the calling process into its own process group and its own session. This is necessary to ensure that two sessions do not contain processes in the same process group.

A full example of process groups is given when we discuss job control in Chapter 15.

When the connection to a terminal is lost, the kernel sends a signal (`SIGHUP`; see Chapter 12 for more information on signals) to the leader of the session containing the terminal's foreground process group, which is usually a shell. This allows the shell to terminate the user's processes unconditionally, notify the processes that the user has logged out (usually, through a `SIGHUP`), or take some other action (or inaction). Although this setup may seem complicated, it lets the session group leader decide how closed terminals should be handled rather than putting that decision in the kernel. This allows system administrators flexibile control over account policies.

Determining the process group is easily done through the `getpgid()` and `getpgrp()` functions.

`pid_t getpgid(pid_t pid)`
: Returns the pgid of process `pid`. If `pid` is 0, the pgid of the current process is returned. No special permissions are needed to use this

call; any process may determine the process group to which any other process belongs.

`pid_t getpgrp(void)`
: Returns the pgid of the current process (equivalent to `getpgid(0)`).

10.6.4 Orphaned Process Groups

The mechanism for how processes are terminated (or allowed to continue) when their session disappears is quite complicated. Imagine a session with multiple process groups in it (Figure 10.1 may help you visualize this). The session is being run on a terminal, and a normal system shell is the session leader.

When the session leader (the shell) exits, the process groups are left in a difficult situation. If they are actively running, they can no longer use stdin or stdout as the terminal has been closed. If they have been suspended, they will probably never run again as the user of that terminal cannot easily restart them, but never running means they will not terminate either.

In this situation, each process group is called an **orphaned process group**. POSIX defines this as a process group whose parent is also a member of that process group, or is not a member of that group's session. This is another way of saying that a process group is not orphaned as long as a process in that group has a parent in the same session but a different group.

While both definitions are complicated, the concept is pretty simple. If a process group is suspended and there is not a process around that can tell it to continue, the process group is orphaned.[18]

When the shell exits, any of its child programs become children of init, but stay in their original session. Assuming that every program in the session is a descendant of the shell, all of the process groups in that session become orphaned.[19] When a process group is orphaned, every process in that

18. Chapter 15 makes all of this much clearer, and it may be worth rereading this section after reading that chapter.
19. But maybe not right away. There may be process groups that contain processes who have parents in other process groups in that session. As the parent-child relationship between processes is a tree, there will eventually be a process group that contained only processes whose parents were the shell, and when that process group exits another one becomes orphaned.

process group is sent a SIGHUP, which normally terminates the program. Programs that have chosen not to terminate on SIGHUP are sent a SIGCONT, which resumes any suspended processes. This sequence terminates most processes and makes sure that any processes that are left are able to run (are not suspended).[20]

Once a process has been orphaned, it is forcibly disassociated from its controlling terminal (to allow a new user to make use of that terminal). If programs that continue running try to access that terminal, those attempts result in errors, with errno set to EIO. The processes remain in the same session, and the session ID is not used for a new process ID until every program in that session has exited.

10.7 Introduction to ladsh

To help illustrate many ideas discussed in this book, we develop a subset of a Unix command shell as the book progresses. At the end of the book, the shell will support

- Simple built-in commands
- Command execution
- I/O redirection (>, |, and so on)
- Job control

The full source code of the final version of this shell, called ladsh4.c, appears in Appendix B. As new features are added to ladsh, the changes to the source code are described in the text. To reduce the number of changes made between the versions, some early versions of the shell are a bit more complicated than they need to be. These extra complications make it easier to develop the shell later in the book, however, so please be patient. Just take those pieces on faith for now; we explain them all to you later in the book.

20. This discussion will make much more sense after you read the chapters on signals (Chapter 12) and job control (Chapter 15).

10.7.1 Running External Programs with `ladsh`

Here is the first (and simplest) version of `ladsh`, called `ladsh1`:

```
 1: /* ladsh1.c */
 2:
 3: #include <ctype.h>
 4: #include <errno.h>
 5: #include <fcntl.h>
 6: #include <signal.h>
 7: #include <stdio.h>
 8: #include <stdlib.h>
 9: #include <string.h>
10: #include <sys/ioctl.h>
11: #include <sys/wait.h>
12: #include <unistd.h>
13:
14: #define MAX_COMMAND_LEN 250     /* max length of a single command
15:                                    string */
16: #define JOB_STATUS_FORMAT "[%d] %-22s %.40s\n"
17:
18: struct jobSet {
19:     struct job * head;      /* head of list of running jobs */
20:     struct job * fg;        /* current foreground job */
21: };
22:
23: struct childProgram {
24:     pid_t pid;              /* 0 if exited */
25:     char ** argv;           /* program name and arguments */
26: };
27:
28: struct job {
29:     int jobId;              /* job number */
30:     int numProgs;           /* total number of programs in job */
31:     int runningProgs;       /* number of programs running */
32:     char * text;            /* name of job */
33:     char * cmdBuf;          /* buffer various argv's point into */
34:     pid_t pgrp;             /* process group ID for the job */
35:     struct childProgram * progs; /* array of programs in job */
36:     struct job * next;      /* to track background commands */
37: };
38:
```

```
39: void freeJob(struct job * cmd) {
40:     int i;
41:
42:     for (i = 0; i < cmd->numProgs; i++) {
43:         free(cmd->progs[i].argv);
44:     }
45:     free(cmd->progs);
46:     if (cmd->text) free(cmd->text);
47:     free(cmd->cmdBuf);
48: }
49:
50: int getCommand(FILE * source, char * command) {
51:     if (source == stdin) {
52:         printf("# ");
53:         fflush(stdout);
54:     }
55:
56:     if (!fgets(command, MAX_COMMAND_LEN, source)) {
57:         if (source == stdin) printf("\n");
58:         return 1;
59:     }
60:
61:     /* remove trailing newline */
62:     command[strlen(command) - 1] = '\0';
63:
64:     return 0;
65: }
66:
67: /* Return cmd->numProgs as 0 if no command is present (e.g. an empty
68:    line). If a valid command is found, commandPtr is set to point to
69:    the beginning of the next command (if the original command had
70:    more than one job associated with it) or NULL if no more
71:    commands are present. */
72: int parseCommand(char ** commandPtr, struct job * job, int * isBg) {
73:     char * command;
74:     char * returnCommand = NULL;
75:     char * src, * buf;
76:     int argc = 0;
77:     int done = 0;
78:     int argvAlloced;
79:     char quote = '\0';
```

```
 80:     int count;
 81:     struct childProgram * prog;
 82:
 83:     /* skip leading white space */
 84:     while (**commandPtr && isspace(**commandPtr)) (*commandPtr)++;
 85:
 86:     /* this handles empty lines and leading '#' characters */
 87:         if (!**commandPtr || (**commandPtr=='#')) {
 88:         job->numProgs = 0;
 89:         *commandPtr = NULL;
 90:         return 0;
 91:     }
 92:
 93:     *isBg = 0;
 94:     job->numProgs = 1;
 95:     job->progs = malloc(sizeof(*job->progs));
 96:
 97:     /* We set the argv elements to point inside of this string. The
 98:        memory is freed by freeJob().
 99:
100:        Getting clean memory relieves us of the task of NULL
101:        terminating things and makes the rest of this look a bit
102:        cleaner (though it is, admittedly, a tad less efficient) */
103:     job->cmdBuf = command = calloc(1, strlen(*commandPtr) + 1);
104:     job->text = NULL;
105:
106:     prog = job->progs;
107:
108:     argvAlloced = 5;
109:     prog->argv = malloc(sizeof(*prog->argv) * argvAlloced);
110:     prog->argv[0] = job->cmdBuf;
111:
112:     buf = command;
113:     src = *commandPtr;
114:     while (*src && !done) {
115:         if (quote == *src) {
116:             quote = '\0';
117:         } else if (quote) {
118:             if (*src == '\\') {
119:                 src++;
120:                 if (!*src) {
```

```
121:                     fprintf(stderr,
122:                             "character expected after \\\n");
123:                     freeJob(job);
124:                     return 1;
125:                 }
126:
127:                 /* in shell, "\'" should yield \' */
128:                 if (*src != quote) *buf++ = '\\';
129:             }
130:             *buf++ = *src;
131:         } else if (isspace(*src)) {
132:             if (*prog->argv[argc]) {
133:                 buf++, argc++;
134:                 /* +1 here leaves room for the NULL which
135:                    ends argv */
136:                 if ((argc + 1) == argvAlloced) {
137:                     argvAlloced += 5;
138:                     prog->argv = realloc(prog->argv,
139:                                 sizeof(*prog->argv) * argvAlloced);
140:                 }
141:                 prog->argv[argc] = buf;
142:             }
143:         } else switch (*src) {
144:         case '"':
145:         case '\'':
146:             quote = *src;
147:             break;
148:
149:         case '#':                           /* comment */
150:             done = 1;
151:             break;
152:
153:         case '&':                           /* background */
154:             *isBg = 1;
155:         case ';':                           /* multiple commands */
156:             done = 1;
157:             returnCommand = *commandPtr + (src - *commandPtr) + 1;
158:             break;
159:
160:         case '\\':
161:             src++;
```

```
162:             if (!*src) {
163:                 freeJob(job);
164:                 fprintf(stderr, "character expected after \\\n");
165:                 return 1;
166:             }
167:             /* fallthrough */
168:         default:
169:             *buf++ = *src;
170:         }
171:
172:         src++;
173:     }
174:
175:     if (*prog->argv[argc]) {
176:         argc++;
177:     }
178:     if (!argc) {
179:         freeJob(job);
180:         return 0;
181:     }
182:     prog->argv[argc] = NULL;
183:
184:     if (!returnCommand) {
185:         job->text = malloc(strlen(*commandPtr) + 1);
186:         strcpy(job->text, *commandPtr);
187:     } else {
188:        /* This leaves any trailing spaces, which is a bit sloppy */
189:
190:         count = returnCommand - *commandPtr;
191:         job->text = malloc(count + 1);
192:         strncpy(job->text, *commandPtr, count);
193:         job->text[count] = '\0';
194:     }
195:
196:     *commandPtr = returnCommand;
197:
198:     return 0;
199: }
200:
201: int runCommand(struct job newJob, struct jobSet * jobList,
202:                int inBg) {
```

```
203:     struct job * job;
204:
205:     /* handle built-ins here -- we don't fork() so we
206:        can't background these very easily */
207:     if (!strcmp(newJob.progs[0].argv[0], "exit")) {
208:         /* this should return a real exit code */
209:         exit(0);
210:     } else if (!strcmp(newJob.progs[0].argv[0], "jobs")) {
211:         for (job = jobList->head; job; job = job->next)
212:             printf(JOB_STATUS_FORMAT, job->jobId, "Running",
213:                     job->text);
214:         return 0;
215:     }
216:
217:     /* we only have one program per child job right now, so this is
218:        easy */
219:     if (!(newJob.progs[0].pid = fork())) {
220:         execvp(newJob.progs[0].argv[0], newJob.progs[0].argv);
221:         fprintf(stderr, "exec() of %s failed: %s\n",
222:                 newJob.progs[0].argv[0],
223:                 strerror(errno));
224:         exit(1);
225:     }
226:
227:     /* put our child in its own process group */
228:     setpgid(newJob.progs[0].pid, newJob.progs[0].pid);
229:
230:     newJob.pgrp = newJob.progs[0].pid;
231:
232:     /* find the ID for the job to use */
233:     newJob.jobId = 1;
234:     for (job = jobList->head; job; job = job->next)
235:         if (job->jobId >= newJob.jobId)
236:             newJob.jobId = job->jobId + 1;
237:
238:     /* add the job to the list of running jobs */
239:     if (!jobList->head) {
240:         job = jobList->head = malloc(sizeof(*job));
241:     } else {
242:         for (job = jobList->head; job->next; job = job->next);
243:         job->next = malloc(sizeof(*job));
```

```
244:         job = job->next;
245:     }
246:
247:     *job = newJob;
248:     job->next = NULL;
249:     job->runningProgs = job->numProgs;
250:
251:     if (inBg) {
252:         /* we don't wait for background jobs to return -- append it
253:            to the list of backgrounded jobs and leave it alone */
254:
255:         printf("[%d] %d\n", job->jobId,
256:                newJob.progs[newJob.numProgs - 1].pid);
257:     } else {
258:         jobList->fg = job;
259:
260:         /* move the new process group into the foreground */
261:
262:         if (tcsetpgrp(0, newJob.pgrp))
263:             perror("tcsetpgrp");
264:     }
265:
266:     return 0;
267: }
268:
269: void removeJob(struct jobSet * jobList, struct job * job) {
270:     struct job * prevJob;
271:
272:     freeJob(job);
273:     if (job == jobList->head) {
274:         jobList->head = job->next;
275:     } else {
276:         prevJob = jobList->head;
277:         while (prevJob->next != job) prevJob = prevJob->next;
278:         prevJob->next = job->next;
279:     }
280:
281:     free(job);
282: }
283:
284: /* Checks to see if any background processes have exited -- if they
```

```
285:    have, figure out why and see if a job has completed */
286: void checkJobs(struct jobSet * jobList) {
287:     struct job * job;
288:     pid_t childpid;
289:     int status;
290:     int progNum;
291:
292:     while ((childpid = waitpid(-1, &status, WNOHANG)) > 0) {
293:         for (job = jobList->head; job; job = job->next) {
294:             progNum = 0;
295:             while (progNum < job->numProgs &&
296:                         job->progs[progNum].pid != childpid)
297:                 progNum++;
298:             if (progNum < job->numProgs) break;
299:         }
300:
301:         job->runningProgs--;
302:         job->progs[progNum].pid = 0;
303:
304:         if (!job->runningProgs) {
305:             printf(JOB_STATUS_FORMAT, job->jobId, "Done",
306:                    job->text);
307:             removeJob(jobList, job);
308:         }
309:     }
310:
311:     if (childpid == -1 && errno != ECHILD)
312:         perror("waitpid");
313: }
314:
315: int main(int argc, const char ** argv) {
316:     char command[MAX_COMMAND_LEN + 1];
317:     char * nextCommand = NULL;
318:     struct jobSet jobList = { NULL, NULL };
319:     struct job newJob;
320:     FILE * input = stdin;
321:     int i;
322:     int status;
323:     int inBg;
324:
325:     if (argc > 2) {
```

```
326:         fprintf(stderr, "unexpected arguments; usage: ladsh1 "
327:                         "<commands>\n");
328:         exit(1);
329:     } else if (argc == 2) {
330:         input = fopen(argv[1], "r");
331:         if (!input) {
332:             perror("fopen");
333:             exit(1);
334:         }
335:     }
336:
337:     /* don't pay any attention to this signal; it just confuses
338:        things and isn't really meant for shells anyway */
339:     signal(SIGTTOU, SIG_IGN);
340:
341:     while (1) {
342:         if (!jobList.fg) {
343:             /* no job is in the foreground */
344:
345:             /* see if any background processes have exited */
346:             checkJobs(&jobList);
347:
348:             if (!nextCommand) {
349:                 if (getCommand(input, command)) break;
350:                 nextCommand = command;
351:             }
352:
353:             if (!parseCommand(&nextCommand, &newJob, &inBg) &&
354:                                 newJob.numProgs) {
355:                 runCommand(newJob, &jobList, inBg);
356:             }
357:         } else {
358:             /* a job is running in the foreground; wait for it */
359:             i = 0;
360:             while (!jobList.fg->progs[i].pid) i++;
361:
362:             waitpid(jobList.fg->progs[i].pid, &status, 0);
363:
364:             jobList.fg->runningProgs--;
365:             jobList.fg->progs[i].pid = 0;
366:
```

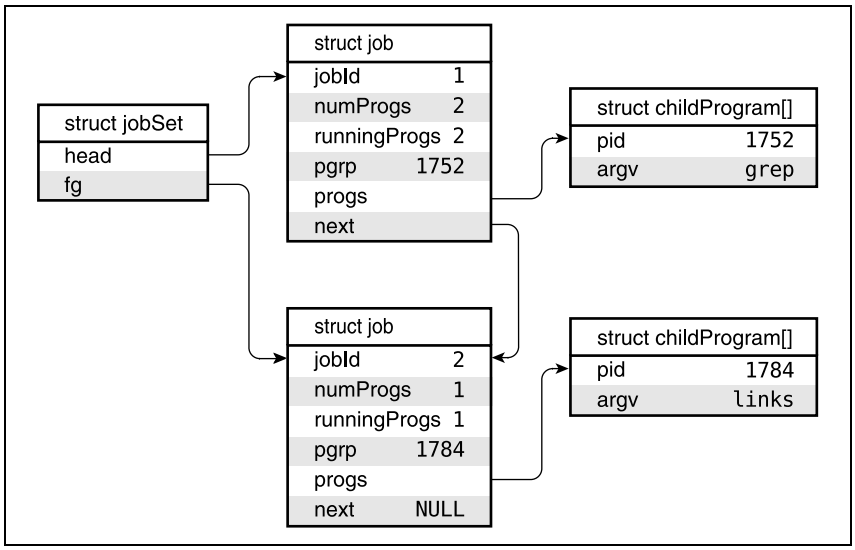

Figure 10.2 Job Data Structures for `ladsh1.c`

```
367:              if (!jobList.fg->runningProgs) {
368:                  /* child exited */
369:
370:                  removeJob(&jobList, jobList.fg);
371:                  jobList.fg = NULL;
372:
373:                  /* move the shell to the foreground */
374:                  if (tcsetpgrp(0, getpid()))
375:                      perror("tcsetpgrp");
376:              }
377:          }
378:      }
379:
380:      return 0;
381: }
```

This version does nothing more than run external programs with arguments, support # comments (everything after a # is ignored), and allow programs to be run in the background. It works as a shell script interpreter for simple scripts written using the `#!` notation, but it does not do much else. It is designed to mimic the usual shell interpreter used on Linux systems, although it is necessarily simplified.

First of all, let's look at the data structures it uses. Figure 10.2 illustrates the data structures `ladsh1.c` uses to keep track of child processes it runs, using a shell running `grep` in the background and `links` in the foreground as an example. `struct jobSet` describes a set of jobs that are running. It contains a linked list of jobs and a pointer to the job that is currently running in the foreground. If there is no foreground job, the pointer is `NULL`. ladsh1.c uses `struct jobSet` to keep track of all the jobs that are currently running as background tasks.

`struct childProgram` describes a single program being executed. This is not quite the same as a job; eventually, we will allow each job to be multiple programs tied together with pipes. For each child program, `ladsh` keeps track of the child's pid, the program that was run, and the command-line arguments. The first element of `argv`, `argv[0]`, holds the name of the program that was run, which is also passed as the first argument to the child.

Multiple commands are tied into a single job by `struct job`. Each job has a job ID unique within the shell, an arbitrary number of programs that constitute the job (stored through `progs`, a pointer to an array of `struct childProgram`), and a pointer to another (next) job, which allows the jobs to be tied together in a linked list (which `struct jobSet` relies on). The job also keeps track of how many individual programs originally comprised the job and how many of those programs are still running (as all of a job's component processes may not quit at the same time). The remaining two members, `text` and `cmdBuf`, are used as buffers that contain various strings used in the `struct childProgram` structures contained by the job.

Much of `struct jobSet` is made up of dynamically allocated memory that should be freed when the job is complete. The first function in ladsh1.c, `freeJob()`, frees the memory used by a job.

The next function, `getCommand()`, reads a command from the user and returns the string. If the commands are being read from a file, no prompting is done (which is why the code compares the input file stream against `stdin`).

`parseCommand()` breaks a command string into a `struct job` for `ladsh` to use. The first argument is a pointer to a pointer to the command. If there are multiple commands in the string, this pointer is advanced to the beginning of the second command. It is set to `NULL` once the final command

from the string has been parsed. This allows parseCommand() to parse only a single command on each invocation and allows the caller to easily parse the entire string through multiple calls. Note that multiple programs piped together do not count as separate commands—only programs separated by ; or & are independent of each other. As parseCommand() is simply an exercise in string parsing, we shall not go over it in any detail.

The runCommand() function is responsible for running a single job. It takes a struct job describing the job to run, the list of jobs that are currently running, and a flag telling it whether the job should be run in the foreground or the background.

For now, ladsh does not support pipes, so each job may consist of only a single program (although much of the infrastructure for supporting pipes is already present in ladsh1.c). If the user runs exit, we exit the program immediately. This is an example of a **built-in** command that must be run by the shell itself to get the proper behavior. Another built-in, jobs, is also implemented here.

If the command is not a built-in, we need to execute a child command. As each job can consist only of a single program (until we implement pipes), this is pretty straightforward.

```
219:     if (!(newJob.progs[0].pid = fork())) {
220:         execvp(newJob.progs[0].argv[0], newJob.progs[0].argv);
221:         fprintf(stderr, "exec() of %s failed: %s\n",
222:                 newJob.progs[0].argv[0],
223:                 strerror(errno));
224:         exit(1);
225:     }
```

First of all, we fork() off the child process. The parent stores the child's pid in newJob.progs[0].pid, whereas the child process places a 0 there (remember, the parent and child have different memory images, although they are initially filled with the same values). This results in the child going into the body of the if statement while the parent skips the body. The child immediately runs the new program via execvp(). If the execvp() call fails, the child prints an error message and then exits. That is all that is necessary for spawning a simple child process.

After forking the child, the parent places the child into its own process group and records the job in the list of running jobs. If the process is meant to run in the foreground, the parent makes the new process group the foreground process group for the shell's controlling terminal.

The next function, `checkJobs()`, looks for background jobs that have exited and cleans up the list of running jobs as appropriate. For each process that has exited (remember `waitpid()` returns only information on exited processes unless `WUNTRACED` is specified), the shell

1. Finds the job the process was a part of
2. Marks the program as completed (by setting the stored pid for the program to 0) and reduces the number of running programs for the job by one

If the job containing the deceased process has no more running processes, which is always the case in this version of `ladsh`, the shell prints a message telling the user the process completed and removes the job from the list of background processes.

The `main()` routine for ladsh1.c controls the shell's execution flow. If an argument was passed to the shell when it was invoked, it treats that as a file name and reads subsequent commands from that file. Otherwise, `stdin` is used as the source of commands. The program then ignores the `SIGTTOU` signal. This is a bit of job-control magic that keeps things going smoothly, and it will make sense once you get to Chapter 15. Job control is not fully implemented, however, and when you are experimenting with ladsh1.c do not try job-control activities (especially those activities that involve suspending programs ladsh1.c has run); they simply will not work. A complete job-control implementation is added in Chapter 15; the setup here is only skeletal.

The remainder of `main()` is the main loop of the program. There is no exit condition for this loop; the program ends by calling `exit()` inside the `runCommand()` function.

The `nextCommand` variable points to the original (unparsed) string representation of the next command that should be run, or is `NULL` if the next command should be read from the input file, which is usually `stdin`. When

no job is running in the foreground, `ladsh` calls `checkJobs()` to check for background jobs that have exited, reads the next command from the input file if `nextCommand` is `NULL`, and then parses and executes the next command.

When a foreground job is executing, ladsh1.c instead waits for one of the processes in the foreground job to terminate. Once all the processes in the foreground job have exited, the job is removed from the list of running jobs and ladsh1.c reads the next command as described previously.

10.8 Creating Clones

Although `fork()` is the traditional way of creating new processes in Unix, Linux also provides the `clone()` system call, which lets the process being duplicated specify what resources the parent process should share with its children.

```
int clone(int flags);
```

This is not much more than a `fork()`; the only difference is the `flags` parameter. It should be set to the signal that should be sent to the parent process when the child exits (usually, `SIGCHLD`), bitwise OR'ed with any number of the following flags, which are defined in `<sched.h>`:

`CLONE_VM`	The two processes share their virtual memory space (including the stack).
`CLONE_FS`	File-system information (such as the current directory) is shared.
`CLONE_FILES`	Open files are shared.
`CLONE_SIGHAND`	The signal handlers are shared for the two processes.

When two resources are shared, both processes see those resources identically. If the `CLONE_SIGHAND` is specified, when one of the processes changes the signal handler for a particular signal, both processes use the new handler (see Chapter 12 for details on signal handlers). When `CLONE_FILES`

is used, not only is the set of open files shared, but the current location in each file is also shared. The return values for `clone()` are the same as for `fork()`.

If a signal other than `SIGCHLD` is specified to be delivered to the parent on the death of a child process, the `wait()` family of functions will not, by default, return information on those processes. If you would like to get information on such processes, as well as on processes that use the normal `SIGCHLD` mechanism, the `__WCLONE` flag must be OR'ed with the `flags` parameter of the wait call. Although this behavior may seem odd, it allows greater flexibility. If `wait()` returned information on cloned processes, it would be more difficult to build standard thread libraries around `clone()`, because `wait()` would return information on other threads, as well as child processes.

Although it is not recommended that applications use `clone()` directly, numerous user-space libraries are available that use `clone()` and provide a fully POSIX-compatible thread implementation. The glibc library includes `libpthread`, which provides the most popular thread implementation. Several good books on POSIX thread programming are available [Butenhof, 1997] [Nichols, 1996].