9

530128

Process Relationships

9.1 Introduction

We learned in the previous chapter that there are relationships between processes. First, every process has a parent process (the initial kernel-level process is usually its own parent). The parent is notified when the child terminates, and the parent can obtain the child's exit status. We also mentioned process groups when we described the `waitpid` function (Section 8.6) and explained how we can wait for any process in a process group to terminate.

In this chapter, we'll look at process groups in more detail and the concept of sessions that was introduced by POSIX.1. We'll also look at the relationship between the login shell that is invoked for us when we log in and all the processes that we start from our login shell.

It is impossible to describe these relationships without talking about signals, and to talk about signals, we need many of the concepts in this chapter. If you are unfamiliar with the UNIX System signal mechanism, you may want to skim through Chapter 10 at this point.

9.2 Terminal Logins

Let's start by looking at the programs that are executed when we log in to a UNIX system. In early UNIX systems, such as Version 7, users logged in using dumb terminals that were connected to the host with hard-wired connections. The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel. For example, the

common devices on PDP-11s were DH-11s and DZ-11s. A host had a fixed number of these terminal devices, so there was a known upper limit on the number of simultaneous logins.

As bitmapped graphical terminals became available, windowing systems were developed to provide users with new ways to interact with host computers. Applications were developed to create "terminal windows" to emulate character-based terminals, allowing users to interact with hosts in familiar ways (i.e., via the shell command line).

Today, some platforms allow you to start a windowing system after logging in, whereas other platforms automatically start the windowing system for you. In the latter case, you might still have to log in, depending on how the windowing system is configured (some windowing systems can be configured to log you in automatically).

The procedure that we now describe is used to log in to a UNIX system using a terminal. The procedure is similar regardless of the type of terminal we use—it could be a character-based terminal, a graphical terminal emulating a simple character-based terminal, or a graphical terminal running a windowing system.

BSD Terminal Logins

The BSD terminal login procedure has not changed much over the past 35 years. The system administrator creates a file, usually `/etc/ttys`, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the `getty` program. One parameter is the baud rate of the terminal, for example. When the system is bootstrapped, the kernel creates process ID 1, the `init` process, and it is `init` that brings the system up in multiuser mode. The `init` process reads the file `/etc/ttys` and, for every terminal device that allows a login, does a `fork` followed by an `exec` of the program `getty`. This gives us the processes shown in Figure 9.1.

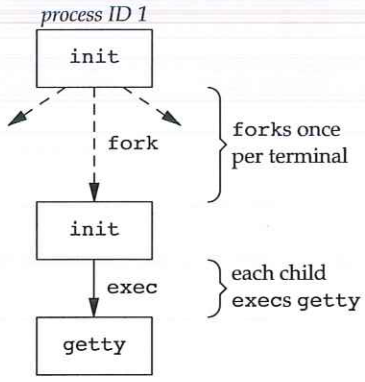

Figure 9.1 Processes invoked by `init` to allow terminal logins

All the processes shown in Figure 9.1 have a real user ID of 0 and an effective user ID of 0 (i.e., they all have superuser privileges). The `init` process also `execs` the `getty` program with an empty environment.

It is getty that calls open for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the open may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then getty outputs something like login: and waits for us to enter our user name. If the terminal supports multiple speeds, getty can detect special characters that tell it to change the terminal's speed (baud rate). Consult your UNIX system manuals for additional details on the getty program and the data files (gettytab) that can drive its actions.

When we enter our user name, getty's job is complete, and it then invokes the login program, similar to

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

(There can be options in the gettytab file to have it invoke other programs, but the default is the login program.) init invokes getty with an empty environment; getty creates an environment for login (the envp argument) with the name of the terminal (something like TERM=foo, where the type of terminal foo is taken from the gettytab file) and any environment strings that are specified in the gettytab. The -p flag to login tells it to preserve the environment that it is passed and to add to that environment, not replace it. Figure 9.2 shows the state of these processes right after login has been invoked.

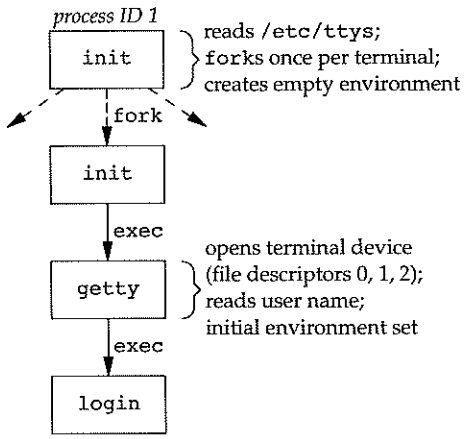

Figure 9.2 State of processes after login has been invoked

All the processes shown in Figure 9.2 have superuser privileges, since the original init process has superuser privileges. The process ID of the bottom three processes in Figure 9.2 is the same, since the process ID does not change across an exec. Also, all the processes other than the original init process have a parent process ID of 1.

The login program does many things. Since it has our user name, it can call getpwnam to fetch our password file entry. Then login calls getpass(3) to display the prompt Password: and read our password (with echoing disabled, of course). It calls crypt(3) to encrypt the password that we entered and compares the encrypted

result to the `pw_passwd` field from our shadow password file entry. If the login attempt fails because of an invalid password (after a few tries), `login` calls `exit` with an argument of 1. This termination will be noticed by the parent (`init`), and it will do another `fork` followed by an `exec` of `getty`, starting the procedure over again for this terminal.

This is the traditional authentication procedure used on UNIX systems. Modern UNIX systems, however, have evolved to support multiple authentication procedures. For example, FreeBSD, Linux, Mac OS X, and Solaris all support a more flexible scheme known as PAM (Pluggable Authentication Modules). PAM allows an administrator to configure the authentication methods to be used to access services that are written to use the PAM library.

If our application needs to verify that a user has the appropriate permission to perform a task, we can either hard code the authentication mechanism in the application or use the PAM library to give us the equivalent functionality. The advantage to using PAM is that administrators can configure different ways to authenticate users for different tasks, based on the local site policies.

If we log in correctly, `login` will

- Change to our home directory (`chdir`)
- Change the ownership of our terminal device (`chown`) so we own it
- Change the access permissions for our terminal device so we have permission to read from and write to it
- Set our group IDs by calling `setgid` and `initgroups`
- Initialize the environment with all the information that `login` has: our home directory (`HOME`), shell (`SHELL`), user name (`USER` and `LOGNAME`), and a default path (`PATH`)
- Change to our user ID (`setuid`) and invoke our login shell, as in

```
execl("/bin/sh", "-sh", (char *)0);
```

The minus sign as the first character of `argv[0]` is a flag to all the shells that indicates they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

The `login` program really does more than we've described here. It optionally prints the message-of-the-day file, checks for new mail, and performs other tasks. In this chapter, we're interested only in the features that we've described.

Recall from our discussion of the `setuid` function in Section 8.11 that since it is called by a superuser process, `setuid` changes all three user IDs: the real user ID, effective user ID, and saved set-user-ID. The call to `setgid` that was done earlier by `login` has the same effect on all three group IDs.

At this point, our login shell is running. Its parent process ID is the original `init` process (process ID 1), so when our login shell terminates, `init` is notified (it is sent a `SIGCHLD` signal) and it starts the whole procedure over again for this terminal. File descriptors 0, 1, and 2 for our login shell are set to the terminal device. Figure 9.3 shows this arrangement.

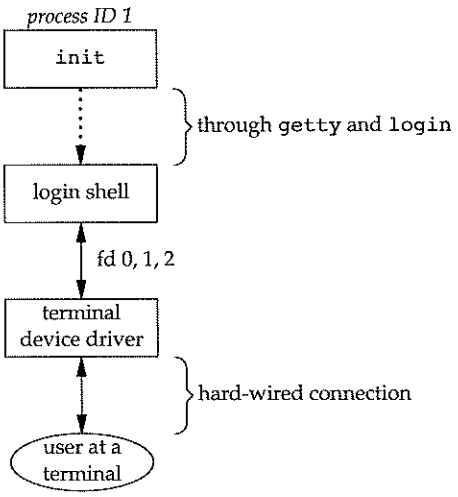

Figure 9.3 Arrangement of processes after everything is set for a terminal login

Our login shell now reads its start-up files (`.profile` for the Bourne shell and Korn shell; `.bash_profile`, `.bash_login`, or `.profile` for the GNU Bourne-again shell; and `.cshrc` and `.login` for the C shell). These start-up files usually change some of the environment variables and add many other variables to the environment. For example, most users set their own `PATH` and often prompt for the actual terminal type (`TERM`). When the start-up files are done, we finally get the shell's prompt and can enter commands.

Mac OS X Terminal Logins

On Mac OS X, the terminal login process follows essentially the same steps as in the BSD login process, since Mac OS X is based in part on FreeBSD. With Mac OS X, however, there are some differences:

- The work of `init` is performed by `launchd`.
- We are presented with a graphical-based login screen from the start.

Linux Terminal Logins

The Linux login procedure is very similar to the BSD procedure. Indeed, the Linux `login` command is derived from the 4.3BSD `login` command. The main difference between the BSD login procedure and the Linux login procedure is in the way the terminal configuration is specified.

Some Linux distributions ship with a version of the `init` program that uses administrative files patterned after System V's `init` file formats. On these systems,

/etc/inittab contains the configuration information specifying the terminal devices for which init should start a getty process.

Other Linux distributions, such as recent Ubuntu distributions, ship with a version of init that is known as "Upstart." It uses configuration files named *.conf that are stored in the /etc/init directory. For example, the specifications for running getty on /dev/tty1 might be found in the file /etc/init/tty1.conf.

Depending on the version of getty in use, the terminal characteristics are specified either on the command line (as with agetty) or in the file /etc/gettydefs (as with mgetty).

Solaris Terminal Logins

Solaris supports two forms of terminal logins: (a) getty style, as described previously for BSD, and (b) ttymon logins, a feature introduced with SVR4. Normally, getty is used for the console, and ttymon is used for other terminal logins.

The ttymon command is part of a larger facility termed SAF, the Service Access Facility. The goal of the SAF was to provide a consistent way to administer services that provide access to a system. (See Chapter 6 of Rago [1993] for more details.) For our purposes, we end up with the same picture as in Figure 9.3, with a different set of steps between init and the login shell. init is the parent of sac (the service access controller), which does a fork and exec of the ttymon program when the system enters multiuser state. The ttymon program monitors all the terminal ports listed in its configuration file and does a fork when we enter our login name. This child of ttymon does an exec of login, and login prompts us for our password. Once this is done, login execs our login shell, and we're at the position shown in Figure 9.3. One difference is that the parent of our login shell is now ttymon, whereas the parent of the login shell from a getty login is init.

9.3 Network Logins

The main (physical) difference between logging in to a system through a serial terminal and logging in to a system through a network is that the connection between the terminal and the computer isn't point-to-point. In this case, login is simply a service available, just like any other network service, such as FTP or SMTP.

With the terminal logins that we described in the previous section, init knows which terminal devices are enabled for logins and spawns a getty process for each device. In the case of network logins, however, all the logins come through the kernel's network interface drivers (e.g., the Ethernet driver), and we don't know ahead of time how many of these will occur. Instead of having a process waiting for each possible login, we now have to wait for a network connection request to arrive.

To allow the same software to process logins over both terminal logins and network logins, a software driver called a *pseudo terminal* is used to emulate the behavior of a serial terminal and map terminal operations to network operations, and vice versa. (In Chapter 19, we'll talk about pseudo terminals in detail.)

BSD Network Logins

In BSD, a single process waits for most network connections: the `inetd` process, sometimes called the *Internet superserver*. In this section, we'll look at the sequence of processes involved in network logins for a BSD system. We are not interested in the detailed network programming aspects of these processes; refer to Stevens, Fenner, and Rudoff [2004] for all the details.

As part of the system start-up, `init` invokes a shell that executes the shell script `/etc/rc`. One of the daemons that is started by this shell script is `inetd`. Once the shell script terminates, the parent process of `inetd` becomes `init`; `inetd` waits for TCP/IP connection requests to arrive at the host. When a connection request arrives for it to handle, `inetd` does a `fork` and `exec` of the appropriate program.

Let's assume that a TCP connection request arrives for the TELNET server. TELNET is a remote login application that uses the TCP protocol. A user on another host (that is connected to the server's host through a network of some form) or on the same host initiates the login by starting the TELNET client:

```
telnet hostname
```

The client opens a TCP connection to *hostname*, and the program that's started on *hostname* is called the TELNET server. The client and the server then exchange data across the TCP connection using the TELNET application protocol. What has happened is that the user who started the client program is now logged in to the server's host. (This assumes, of course, that the user has a valid account on the server's host.) Figure 9.4 shows the sequence of processes involved in executing the TELNET server, called `telnetd`.

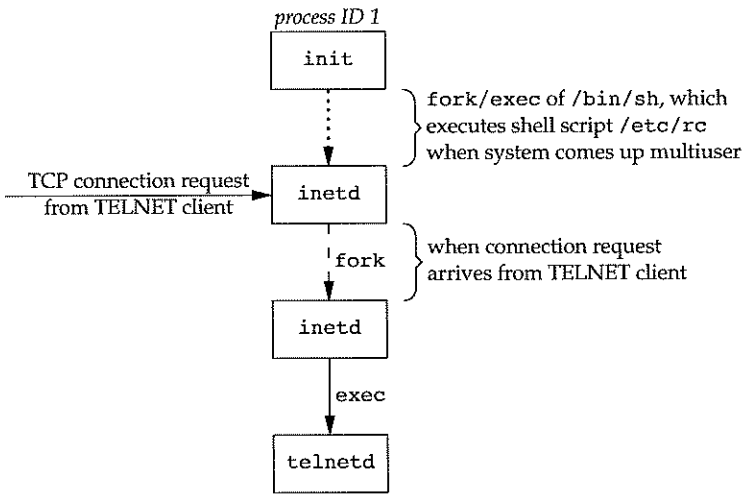

Figure 9.4 Sequence of processes involved in executing TELNET server

The `telnetd` process then opens a pseudo terminal device and splits into two processes using `fork`. The parent handles the communication across the network connection, and the child does an `exec` of the `login` program. The parent and the child are connected through the pseudo terminal. Before doing the `exec`, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, `login` performs the same steps we described in Section 9.2: it changes to our home directory and sets our group IDs, user ID, and our initial environment. Then `login` replaces itself with our login shell by calling `exec`. Figure 9.5 shows the arrangement of the processes at this point.

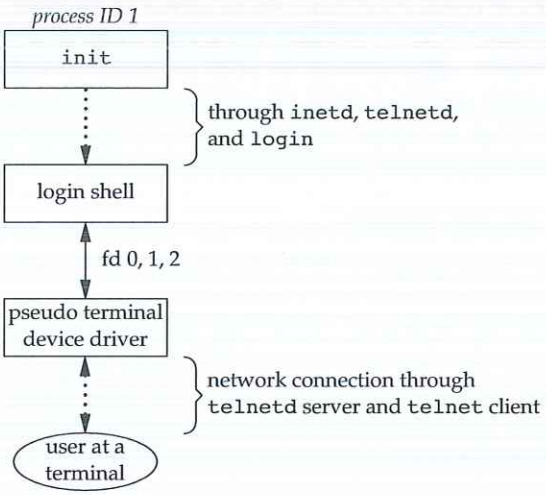

Figure 9.5 Arrangement of processes after everything is set for a network login

Obviously, a lot is going on between the pseudo terminal device driver and the actual user at the terminal. We'll show all the processes involved in this type of arrangement in Chapter 19 when we talk about pseudo terminals in more detail.

The important thing to understand is that whether we log in through a terminal (Figure 9.3) or a network (Figure 9.5), we have a login shell with its standard input, standard output, and standard error connected to either a terminal device or a pseudo terminal device. We'll see in the coming sections that this login shell is the start of a POSIX.1 session, and that the terminal or pseudo terminal is the controlling terminal for the session.

Mac OS X Network Logins

Logging in to a Mac OS X system over a network is identical to logging in to a BSD system, because Mac OS X is based partially on FreeBSD. However, on Mac OS X, the `telnet` daemon is run from `launchd`.

By default, the `telnet` daemon is disabled on Mac OS X (although it can be enabled with the `launchctl`(1) command). The preferred way to perform a network login on Mac OS X is with `ssh`, the secure shell command.

Linux Network Logins

Network logins under Linux are the same as under BSD, except that some distributions use an alternative `inetd` process called the extended Internet services daemon, `xinetd`. The `xinetd` process provides a finer level of control over services it starts compared to `inetd`.

Solaris Network Logins

The scenario for network logins under Solaris is almost identical to the steps under BSD and Linux. An `inetd` server is used that is similar in concept to the BSD version, except that the Solaris version runs as a *restarter* in the Service Management Facility (SMF). A restarter is a daemon that has the responsibility to start and monitor other daemon processes, and restart them if they fail. Although the `inetd` server is started by the master restarter in the SMF, the master restarter is started by `init` and we end up with the same overall picture as in Figure 9.5.

The Solaris Service Management Facility is a framework that manages and monitors system services and provides a way to recover from failures affecting system services. For more details on the Service Management Facility, see Adams [2005] and the Solaris manual pages `smf`(5) and `inetd`(1M).

9.4 Process Groups

In addition to having a process ID, each process belongs to a process group. We'll encounter process groups again when we discuss signals in Chapter 10.

A process group is a collection of one or more processes, usually associated with the same job (job control is discussed in Section 9.8), that can receive signals from the same terminal. Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a `pid_t` data type. The function `getpgrp` returns the process group ID of the calling process.

```
#include <unistd.h>

pid_t getpgrp(void);
```

Returns: process group ID of calling process

In older BSD-derived systems, the `getpgrp` function took a *pid* argument and returned the process group for that process. The Single UNIX Specification defines the `getpgid` function that mimics this behavior.

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

Returns: process group ID if OK, –1 on error

If *pid* is 0, the process group ID of the calling process is returned. Thus

```
getpgid(0);
```

is equivalent to

```
getpgrp();
```

Each process group can have a process group leader. The leader is identified by its process group ID being equal to its process ID.

It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates. This is called the process group lifetime—the period of time that begins when the group is created and ends when the last remaining process leaves the group. The last remaining process in the process group can either terminate or enter some other process group.

A process joins an existing process group or creates a new process group by calling `setpgid`. (In the next section, we'll see that `setsid` also creates a new process group.)

```
#include <unistd.h>

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

Returns: 0 if OK, –1 on error

This function sets the process group ID to *pgid* in the process whose process ID equals *pid*. If the two arguments are equal, the process specified by *pid* becomes a process group leader. If *pid* is 0, the process ID of the caller is used. Also, if *pgid* is 0, the process ID specified by *pid* is used as the process group ID.

A process can set the process group ID of only itself or any of its children. Furthermore, it can't change the process group ID of one of its children after that child has called one of the `exec` functions.

In most job-control shells, this function is called after a `fork` to have the parent set the process group ID of the child, and to have the child set its own process group ID. One of these calls is redundant, but by doing both, we are guaranteed that the child is placed into its own process group before either process assumes that this has happened. If we didn't do this, we would have a race condition, since the child's process group membership would depend on which process executes first.

When we discuss signals, we'll see how we can send a signal to either a single process (identified by its process ID) or a process group (identified by its process group ID). Similarly, the `waitpid` function from Section 8.6 lets us wait for either a single process or one process from a specified process group.

9.5 Sessions

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure 9.6. Here we have three process groups in a single session.

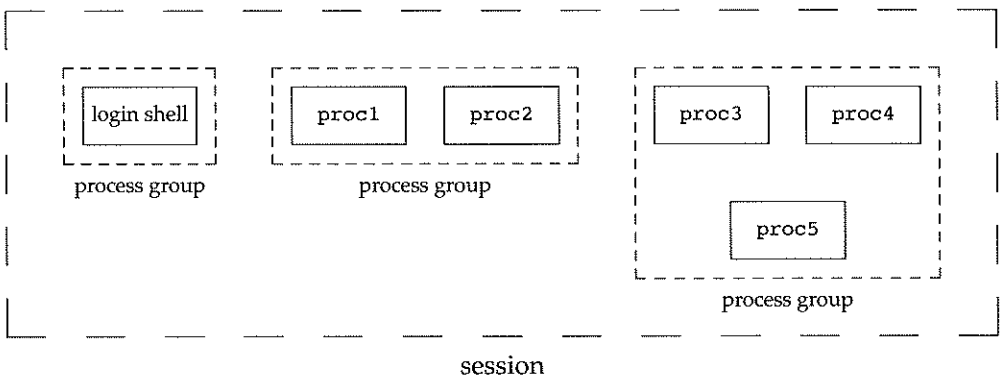

Figure 9.6 Arrangement of processes into process groups and sessions

The processes in a process group are usually placed there by a shell pipeline. For example, the arrangement shown in Figure 9.6 could have been generated by shell commands of the form

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

A process establishes a new session by calling the `setsid` function.

```
#include <unistd.h>

pid_t setsid(void);
```

Returns: process group ID if OK, –1 on error

If the calling process is not a process group leader, this function creates a new session. Three things happen.

1. The process becomes the *session leader* of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
2. The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
3. The process has no controlling terminal. (We'll discuss controlling terminals in the next section.) If the process had a controlling terminal before calling `setsid`, that association is broken.

This function returns an error if the caller is already a process group leader. To ensure this is not the case, the usual practice is to call `fork` and have the parent terminate and the child continue. We are guaranteed that the child is not a process group leader, because the process group ID of the parent is inherited by the child, but the child gets a new process ID. Hence, it is impossible for the child's process ID to equal its inherited process group ID.

The Single UNIX Specification talks only about a "session leader"; there is no "session ID" similar to a process ID or a process group ID. Obviously, a session leader is a single process that has a unique process ID, so we could talk about a session ID that is the process ID of the session leader. This concept of a session ID was introduced in SVR4. Historically, BSD-based systems didn't support this notion, but have since been updated to include it. The `getsid` function returns the process group ID of a process's session leader.

> Some implementations, such as Solaris, join with the Single UNIX Specification in the practice of avoiding the use of the phrase "session ID," opting instead to refer to this as the "process group ID of the session leader." The two are equivalent, since the session leader is always the leader of a process group.

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Returns: session leader's process group ID if OK, –1 on error

If *pid* is 0, `getsid` returns the process group ID of the calling process's session leader. For security reasons, some implementations may restrict the calling process from obtaining the process group ID of the session leader if *pid* doesn't belong to the same session as the caller.

9.6 Controlling Terminal

Sessions and process groups have a few other characteristics.

- A session can have a single *controlling terminal*. This is usually the terminal device (in the case of a terminal login) or pseudo terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the *controlling process*.
- The process groups within a session can be divided into a single *foreground process group* and one or more *background process groups*.
- If a session has a controlling terminal, it has a single foreground process group and all other process groups in the session are background process groups.
- Whenever we press the terminal's interrupt key (often DELETE or Control-C), the interrupt signal is sent to all processes in the foreground process group.

- Whenever we press the terminal's quit key (often Control-backslash), the quit signal is sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

These characteristics are shown in Figure 9.7.

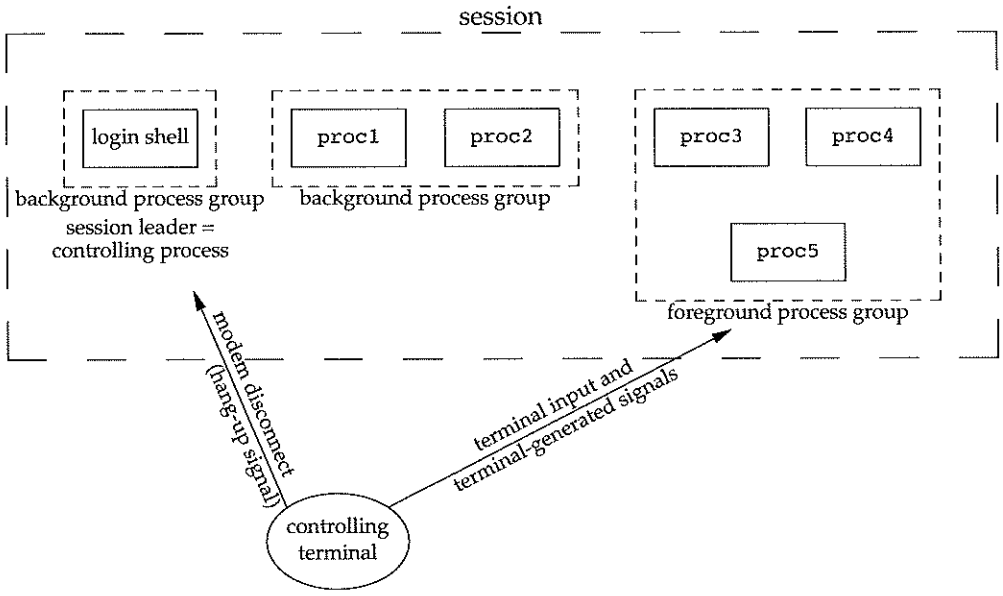

Figure 9.7 Process groups and sessions showing controlling terminal

Usually, we don't have to worry about the controlling terminal; it is established automatically when we log in.

POSIX.1 leaves the choice of the mechanism used to allocate a controlling terminal up to each individual implementation. We'll show the actual steps in Section 19.4.

Systems derived from UNIX System V allocate the controlling terminal for a session when the session leader opens the first terminal device that is not already associated with a session, as long as the call to `open` does not specify the `O_NOCTTY` flag (Section 3.3).

BSD-based systems allocate the controlling terminal for a session when the session leader calls `ioctl` with a *request* argument of `TIOCSCTTY` (the third argument is a null pointer). The session cannot already have a controlling terminal for this call to succeed. (Normally, this call to `ioctl` follows a call to `setsid`, which guarantees that the process is a session leader without a controlling terminal.) The POSIX.1 `O_NOCTTY` flag to `open` is not used by BSD-based systems, except in compatibility-mode support for other systems.

Figure 9.8 summarizes the way each platform discussed in this book allocates a controlling terminal. Note that although Mac OS X 10.6.8 is derived from BSD, it behaves like System V when allocating a controlling terminal.

Method	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
open without `O_NOCTTY`		•	•	•
`TIOCSCTTY` `ioctl` command	•	•	•	•

Figure 9.8 How various implementations allocate controlling terminals

There are times when a program wants to talk to the controlling terminal, regardless of whether the standard input or standard output is redirected. The way a program guarantees that it is talking to the controlling terminal is to open the file `/dev/tty`. This special file is a synonym within the kernel for the controlling terminal. Naturally, if the program doesn't have a controlling terminal, the open of this device will fail.

The classic example is the `getpass`(3) function, which reads a password (with terminal echoing turned off, of course). This function is called by the `crypt`(1) program and can be used in a pipeline. For example,

```
crypt < salaries | lpr
```

decrypts the file `salaries` and pipes the output to the print spooler. Because `crypt` reads its input file on its standard input, the standard input can't be used to enter the password. Also, `crypt` is designed so that we have to enter the encryption password each time we run the program, to prevent us from saving the password in a file (which could be a security hole).

There are known ways to break the encoding used by the `crypt` program. See Garfinkel et al. [2003] for more details on encrypting files.

9.7 tcgetpgrp, tcsetpgrp, and tcgetsid Functions

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals (Figure 9.7).

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);
```

Returns: process group ID of foreground process group if OK, –1 on error

```
int tcsetpgrp(int fd, pid_t pgrpid);
```

Returns: 0 if OK, –1 on error

The function `tcgetpgrp` returns the process group ID of the foreground process group associated with the terminal open on *fd*.

If the process has a controlling terminal, the process can call `tcsetpgrp` to set the foreground process group ID to *pgrpid*. The value of *pgrpid* must be the process group ID of a process group in the same session, and *fd* must refer to the controlling terminal of the session.

Most applications don't call these two functions directly. Instead, the functions are normally called by job-control shells.

The `tcgetsid` function allows an application to obtain the process group ID for the session leader given a file descriptor for the controlling TTY.

```
#include <termios.h>

pid_t tcgetsid(int fd);
```

Returns: session leader's process group ID if OK, –1 on error

Applications that need to manage controlling terminals can use `tcgetsid` to identify the session ID of the controlling terminal's session leader (which is equivalent to the session leader's process group ID).

9.8 Job Control

Job control is a feature that was added to BSD around 1980. This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are run in the background. Job control requires three forms of support:

1. A shell that supports job control
2. The terminal driver in the kernel must support job control
3. The kernel must support certain job-control signals

> SVR3 provided a different form of job control called *shell layers*. The BSD form of job control, however, was selected by POSIX.1 and is what we describe here. In earlier versions of the standard, job control support was optional, but POSIX.1 now requires platforms to support it.

From our perspective, when using job control from a shell, we can start a job in either the foreground or the background. A job is simply a collection of processes, often a pipeline of processes. For example,

```
vi main.c
```

starts a job consisting of one process in the foreground. The commands

```
pr *.c | lpr &
make all &
```

start two jobs in the background. All the processes invoked by these background jobs are in the background.

As we said, to use the features provided by job control, we need to use a shell that supports job control. With older systems, it was simple to say which shells supported job control and which didn't. The C shell supported job control, the Bourne shell didn't, and it was an option with the Korn shell, depending on whether the host supported job control. But the C shell has been ported to systems (e.g., earlier versions of System V) that don't support job control, and the SVR4 Bourne shell, when invoked by the name `jsh` instead of `sh`, supports job control. The Korn shell continues to support job control

if the host does. The Bourne-again shell also supports job control. We'll just talk generically about a shell that supports job control, versus one that doesn't, when the difference between the various shells doesn't matter.

When we start a background job, the shell assigns it a job identifier and prints one or more of the process IDs. The following script shows how the Korn shell handles this:

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                                   just press RETURN
[2] +  Done          pr *.c | lpr &
[1] +  Done          make all > Make.out &
```

The `make` is job number 1 and the starting process ID is 1475. The next pipeline is job number 2 and the process ID of the first process is 1490. When the jobs are done and we press RETURN, the shell tells us that the jobs are complete. The reason we have to press RETURN is to have the shell print its prompt. The shell doesn't print the changed status of background jobs at any random time—only right before it prints its prompt, to let us enter a new command line. If the shell didn't do this, it could produce output while we were entering an input line.

The interaction with the terminal driver arises because a special terminal character affects the foreground job: the suspend key (typically Control-Z). Entering this character causes the terminal driver to send the `SIGTSTP` signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver looks for three special characters, which generate signals to the foreground process group.

- The interrupt character (typically DELETE or Control-C) generates `SIGINT`.
- The quit character (typically Control-backslash) generates `SIGQUIT`.
- The suspend character (typically Control-Z) generates `SIGTSTP`.

In Chapter 18, we'll see how we can change these three characters to be any characters we choose and how we can disable the terminal driver's processing of these special characters.

Another job control condition can arise that must be handled by the terminal driver. Since we can have a foreground job and one or more background jobs, which of these receives the characters that we enter at the terminal? Only the foreground job receives terminal input. It is not an error for a background job to try to read from the terminal, but the terminal driver detects this and sends a special signal to the background job: `SIGTTIN`. This signal normally stops the background job; by using the shell, we are notified of this event and can bring the job into the foreground so that it can read from the terminal. The following example demonstrates this:

```
$ cat > temp.foo &          start in background, but it'll read from standard input
[1]     1681
$                           we press RETURN
[1] + Stopped (SIGTTIN)       cat > temp.foo &
```

```
$ fg %1                    bring job number 1 into the foreground
cat > temp.foo             the shell tells us which job is now in the foreground
hello, world               enter one line
^D                         type the end-of-file character
$ cat temp.foo             check that the one line was put into the file
hello, world
```

Note that this example doesn't work on Mac OS X 10.6.8. When we try to bring the `cat` command into the foreground, the `read` fails with `errno` set to `EINTR`. Since Mac OS X is based on FreeBSD, and FreeBSD works as expected, this must be a bug in Mac OS X.

The shell starts the `cat` process in the background, but when `cat` tries to read its standard input (the controlling terminal), the terminal driver, knowing that it is a background job, sends the `SIGTTIN` signal to the background job. The shell detects this change in status of its child (recall our discussion of the `wait` and `waitpid` function in Section 8.6) and tells us that the job has been stopped. We then move the stopped job into the foreground with the shell's `fg` command. (Refer to the manual page for the shell that you are using for all the details on its job control commands, such as `fg` and `bg`, and the various ways to identify the different jobs.) Doing this causes the shell to place the job into the foreground process group (`tcsetpgrp`) and send the continue signal (`SIGCONT`) to the process group. Since it is now in the foreground process group, the job can read from the controlling terminal.

What happens if a background job sends its output to the controlling terminal? This is an option that we can allow or disallow. Normally, we use the `stty`(1) command to change this option. (We'll see in Chapter 18 how we can change this option from a program.) The following example shows how this works:

```
$ cat temp.foo &           execute in background
[1]     1719
$ hello, world             the output from the background job appears after the prompt
                           we press RETURN
[1] +  Done        cat temp.foo &
$ stty tostop              disable ability of background jobs to output to controlling terminal
$ cat temp.foo &           try it again in the background
[1]     1721
$                          we press RETURN and find the job is stopped
[1] + Stopped(SIGTTOU)        cat temp.foo &
$ fg %1                    resume stopped job in the foreground
cat temp.foo               the shell tells us which job is now in the foreground
hello, world               and here is its output
```

When we disallow background jobs from writing to the controlling terminal, `cat` will block when it tries to write to its standard output, because the terminal driver identifies the write as coming from a background process and sends the job the `SIGTTOU` signal. As with the previous example, when we use the shell's `fg` command to bring the job into the foreground, the job completes.

Figure 9.9 summarizes some of the features of job control that we've been describing. The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process

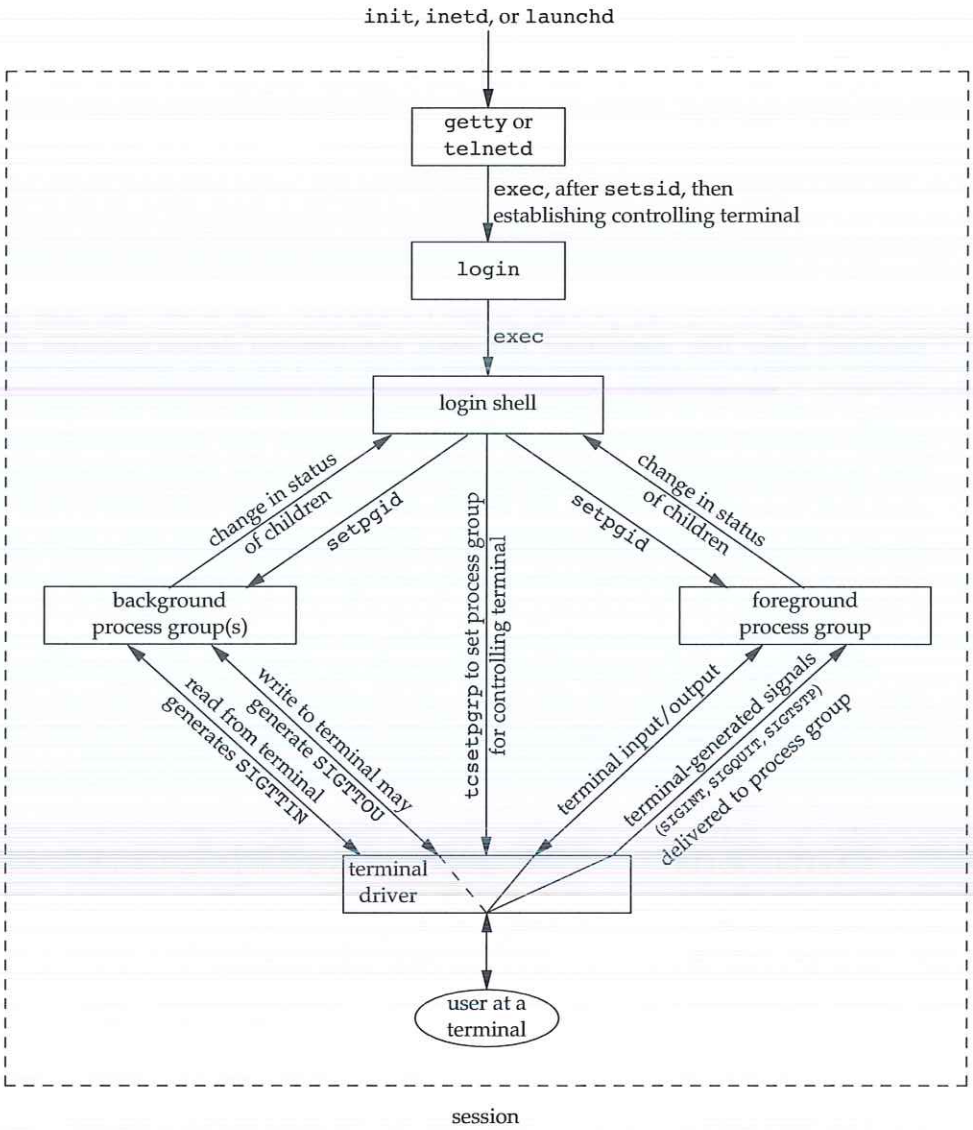

Figure 9.9 Summary of job control features with foreground and background jobs, and terminal driver

group to the actual terminal. The dashed line corresponding to the SIGTTOU signal means that whether the output from a process in the background process group appears on the terminal is an option.

Is job control necessary or desirable? Job control was originally designed and implemented before windowing terminals were widespread. Some people claim that a well-designed windowing system removes any need for job control. Some complain that the implementation of job control—requiring support from the kernel, the terminal

driver, the shell, and some applications—is a hack. Some use job control with a windowing system, claiming a need for both. Regardless of your opinion, job control is a required feature of POSIX.1.

9.9 Shell Execution of Programs

Let's examine how the shells execute programs and how this relates to the concepts of process groups, controlling terminals, and sessions. To do this, we'll use the `ps` command again.

First, we'll use a shell that doesn't support job control—the classic Bourne shell running on Solaris. If we execute

```
ps -o pid,ppid,pgid,sid,comm
```

the output is

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1774   949   949   949 ps
```

The parent of the `ps` command is the shell, which we would expect. Both the shell and the `ps` command are in the same session and foreground process group (949). We say that 949 is the foreground process group because that is what you get when you execute a command with a shell that doesn't support job control.

> Some platforms support an option to have the `ps`(1) command print the process group ID associated with the session's controlling terminal. This value would be shown under the TPGID column. Unfortunately, the output of the `ps` command often differs among versions of the UNIX System. For example, Solaris 10 doesn't support this option. Under FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, the command
>
> ```
> ps -o pid,ppid,pgid,sid,tpgid,comm
> ```
>
> prints exactly the information we want.
>
> Note that it is misleading to associate a process with a terminal process group ID (the TPGID column). A process does not have a terminal process control group. A process belongs to a process group, and the process group belongs to a session. The session may or may not have a controlling terminal. If the session does have a controlling terminal, then the terminal device knows the process group ID of the foreground process. This value can be set in the terminal driver with the `tcsetpgrp` function, as we show in Figure 9.9. The foreground process group ID is an attribute of the terminal, not the process. This value from the terminal device driver is what `ps` prints as the TPGID. If it finds that the session doesn't have a controlling terminal, `ps` prints either 0 or –1, depending on the platform.

If we execute the command in the background,

```
ps -o pid,ppid,pgid,sid,comm &
```

the only value that changes is the process ID of the command:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1812   949   949   949 ps
```

This shell doesn't know about job control, so the background job is not put into its own process group and the controlling terminal isn't taken away from the background job.

Now let's look at how the Bourne shell handles a pipeline. When we execute

```
ps -o pid,ppid,pgid,sid,comm | cat1
```

the output is

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1823   949   949   949 cat1
 1824  1823   949   949 ps
```

(The program `cat1` is just a copy of the standard `cat` program, with a different name. We have another copy of `cat` with the name `cat2`, which we'll use later in this section. When we have two copies of `cat` in a pipeline, the different names let us differentiate between the two programs.) Note that the last process in the pipeline is the child of the shell and that the first process in the pipeline is a child of the last process. It appears that the shell `forks` a copy of itself and that this copy then `forks` to make each of the previous processes in the pipeline.

If we execute the pipeline in the background,

```
ps -o pid,ppid,pgid,sid,comm | cat1 &
```

only the process IDs change. Since the shell doesn't handle job control, the process group ID of the background processes remains 949, as does the process group ID of the session.

What happens in this case if a background process tries to read from its controlling terminal? For example, suppose that we execute

```
cat > temp.foo &
```

With job control, this is handled by placing the background job into a background process group, which causes the signal `SIGTTIN` to be generated if the background job tries to read from the controlling terminal. The way this is handled without job control is that the shell automatically redirects the standard input of a background process to `/dev/null`, if the process doesn't redirect standard input itself. A read from `/dev/null` generates an end of file. This means that our background `cat` process immediately reads an end of file and terminates normally.

The previous paragraph adequately handles the case of a background process accessing the controlling terminal through its standard input, but what happens if a background process specifically opens `/dev/tty` and reads from the controlling terminal? The answer is "It depends," but the result is probably not what we want. For example,

```
crypt < salaries | lpr &
```

is such a pipeline. We run it in the background, but the `crypt` program opens `/dev/tty`, changes the terminal characteristics (to disable echoing), reads from the device, and resets the terminal characteristics. When we execute this background pipeline, the prompt `Password:` from `crypt` is printed on the terminal, but what we enter (the encryption password) is read by the shell, which tries to execute a command

of that name. The next line we enter to the shell is taken as the password, and the file is not decrypted correctly, sending junk to the printer. Here we have two processes trying to read from the same device at the same time, and the result depends on the system. Job control, as we described earlier, handles this multiplexing of a single terminal between multiple processes in a better fashion.

Returning to our Bourne shell example, if we execute three processes in the pipeline, we can examine the process control used by this shell:

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

This pipeline generates the following output:

```
 PID  PPID  PGID   SID COMMAND
 949   947   949   949 sh
1988   949   949   949 cat2
1989  1988   949   949 ps
1990  1988   949   949 cat1
```

Don't be alarmed if the output on your system doesn't show the proper command names. Sometimes you might get results such as

```
 PID  PPID  PGID   SID COMMAND
 949   947   949   949 sh
1831   949   949   949 sh
1832  1831   949   949 ps
1833  1831   949   949 sh
```

What's happening here is that the `ps` process is racing with the shell, which is forking and executing the `cat` commands. In this case, the shell hasn't yet completed the call to `exec` when `ps` has obtained the list of processes to print.

Again, the last process in the pipeline is the child of the shell, and all previous processes in the pipeline are children of the last process. Figure 9.10 shows what is happening.

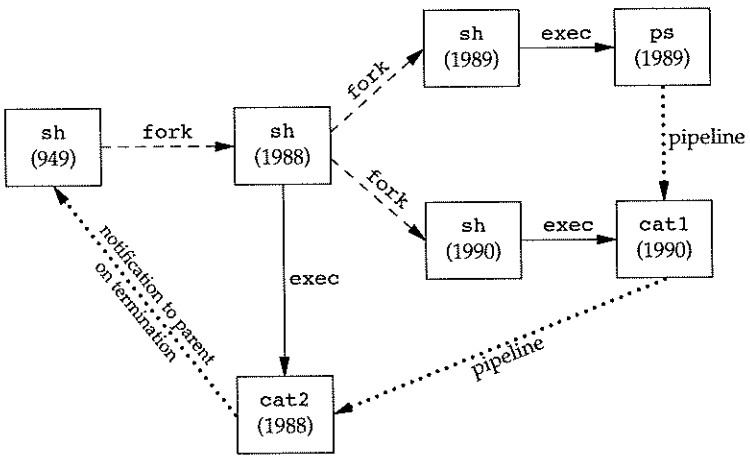

Figure 9.10 Processes in the pipeline `ps | cat1 | cat2` when invoked by Bourne shell

Since the last process in the pipeline is the child of the login shell, the shell is notified when that process (`cat2`) terminates.

Now let's examine the same examples using a job-control shell running on Linux. This shows the way these shells handle background jobs. We'll use the Bourne-again shell in this example; the results with other job-control shells are almost identical.

```
ps -o pid,ppid,pgid,sid,tpgid,comm
```

gives us

```
  PID  PPID  PGID   SID TPGID COMMAND
 2837  2818  2837  2837  5796 bash
 5796  2837  5796  2837  5796 ps
```

(Starting with this example, we show the foreground process group in a **bolder font**.) We immediately see a difference from our Bourne shell example. The Bourne-again shell places the foreground job (`ps`) into its own process group (5796). The `ps` command is the process group leader and the only process in this process group. Furthermore, this process group is the foreground process group, since it has the controlling terminal. Our login shell is a background process group while the `ps` command executes. Note, however, that both process groups, 2837 and 5796, are members of the same session. Indeed, we'll see that the session never changes through our examples in this section.

Executing this process in the background,

```
ps -o pid,ppid,pgid,sid,tpgid,comm &
```

gives us

```
  PID  PPID  PGID   SID TPGID COMMAND
 2837  2818  2837  2837  2837 bash
 5797  2837  5797  2837  2837 ps
```

Again, the `ps` command is placed into its own process group, but this time the process group (5797) is no longer the foreground process group—it is a background process group. The TPGID of 2837 indicates that the foreground process group is our login shell.

Executing two processes in a pipeline, as in

```
ps -o pid,ppid,pgid,sid,tpgid,comm | cat1
```

gives us

```
  PID  PPID  PGID   SID TPGID COMMAND
 2837  2818  2837  2837  5799 bash
 5799  2837  5799  2837  5799 ps
 5800  2837  5799  2837  5799 cat1
```

Both processes, `ps` and `cat1`, are placed into a new process group (5799), and this is the foreground process group. We can also see another difference between this example and the similar Bourne shell example. The Bourne shell created the last process in the pipeline first, and this final process was the parent of the first process. Here, the Bourne-again shell is the parent of both processes. If we execute this pipeline in the background,

```
ps -o pid,ppid,pgid,sid,tpgid,comm | cat1 &
```

the results are similar, but now `ps` and `cat1` are placed in the same background process group:

```
 PID  PPID  PGID   SID TPGID COMMAND
2837  2818  2837  2837  2837 bash
5801  2837  5801  2837  2837 ps
5802  2837  5801  2837  2837 cat1
```

Note that the order in which a shell creates processes can differ depending on the particular shell in use.

9.10 Orphaned Process Groups

We've mentioned that a process whose parent terminates is called an orphan and is inherited by the `init` process. We now look at entire process groups that can be orphaned and see how POSIX.1 handles this situation.

Example

Consider a process that `forks` a child and then terminates. Although this is nothing abnormal (it happens all the time), what happens if the child is stopped (using job control) when the parent terminates? How will the child ever be continued, and does the child know that it has been orphaned? Figure 9.11 shows this situation: the parent process has `forked` a child that stops, and the parent is about to exit.

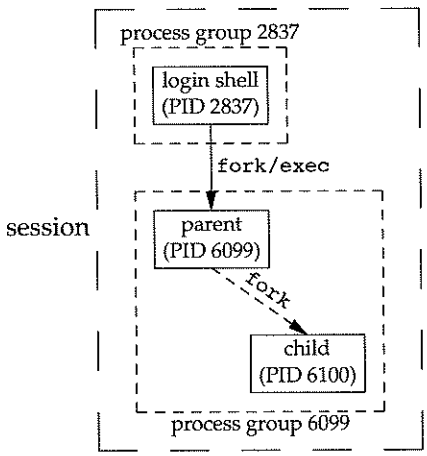

Figure 9.11 Example of a process group about to be orphaned

The program that creates this situation is shown in Figure 9.12. This program has some new features. Here, we are assuming a job-control shell. Recall from the previous section that the shell places the foreground process into its own process group (6099 in

```
#include "apue.h"
#include <errno.h>

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %ld\n", (long)getpid());
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %ld, ppid = %ld, pgrp = %ld, tpgrp = %ld\n",
        name, (long)getpid(), (long)getppid(), (long)getpgrp(),
        (long)tcgetpgrp(STDIN_FILENO));
    fflush(stdout);
}

int
main(void)
{
    char    c;
    pid_t   pid;

    pr_ids("parent");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        sleep(5);       /* sleep to let child stop itself */
    } else {            /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);    /* establish signal handler */
        kill(getpid(), SIGTSTP);    /* stop ourself */
        pr_ids("child");    /* prints only if we're continued */
        if (read(STDIN_FILENO, &c, 1) != 1)
            printf("read error %d on controlling TTY\n", errno);
    }
    exit(0);
}
```

Figure 9.12 Creating an orphaned process group

this example) and that the shell stays in its own process group (2837). The child inherits the process group of its parent (6099). After the `fork`,

- The parent sleeps for 5 seconds. This is our (imperfect) way of letting the child execute before the parent terminates.
- The child establishes a signal handler for the hang-up signal (`SIGHUP`) so we can see whether it is sent to the child. (We discuss signal handlers in Chapter 10.)
- The child sends itself the stop signal (`SIGTSTP`) with the `kill` function. This stops the child, similar to our stopping a foreground job with our terminal's suspend character (Control-Z).

- When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, which is the `init` process ID.
- At this point, the child is now a member of an *orphaned process group*. The POSIX.1 definition of an orphaned process group is one in which the parent of every member is either itself a member of the group or is not a member of the group's session. Another way of saying this is that the process group is not orphaned as long as a process in the group has a parent in a different process group but in the same session. If the process group is not orphaned, there is a chance that one of those parents in a different process group but in the same session will restart a stopped process in the process group that is not orphaned. Here, the parent of every process in the group (e.g., process 1 is the parent of process 6100) belongs to another session.
- Since the process group is orphaned when the parent terminates, and the process group contains a stopped process, POSIX.1 requires that every process in the newly orphaned process group be sent the hang-up signal (`SIGHUP`) followed by the continue signal (`SIGCONT`).
- This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, so we have to provide a signal handler to catch the signal. We therefore expect the `printf` in the `sig_hup` function to appear before the `printf` in the `pr_ids` function.

Here is the output from the program shown in Figure 9.12:

```
$ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error 5 on controlling TTY
```

Note that our shell prompt appears with the output from the child, since two processes—our login shell and the child—are writing to the terminal. As we expect, the parent process ID of the child has become 1.

After calling `pr_ids` in the child, the program tries to read from standard input. As we saw earlier in this chapter, when a process in a background process group tries to read from its controlling terminal, `SIGTTIN` is generated for the background process group. But here we have an orphaned process group; if the kernel were to stop it with this signal, the processes in the process group would probably never be continued. POSIX.1 specifies that the `read` is to return an error with `errno` set to `EIO` (whose value is 5 on this system) in this situation.

Finally, note that our child was placed in a background process group when the parent terminated, since the parent was executed as a foreground job by the shell. □

We'll see another example of orphaned process groups in Section 19.5 with the `pty` program.

9.11 FreeBSD Implementation

Having talked about the various attributes of a process, process group, session, and controlling terminal, it's worth looking at how all this can be implemented. We'll look briefly at the implementation used by FreeBSD. Some details of the SVR4 implementation of these features can be found in Williams [1989]. Figure 9.13 shows the various data structures used by FreeBSD.

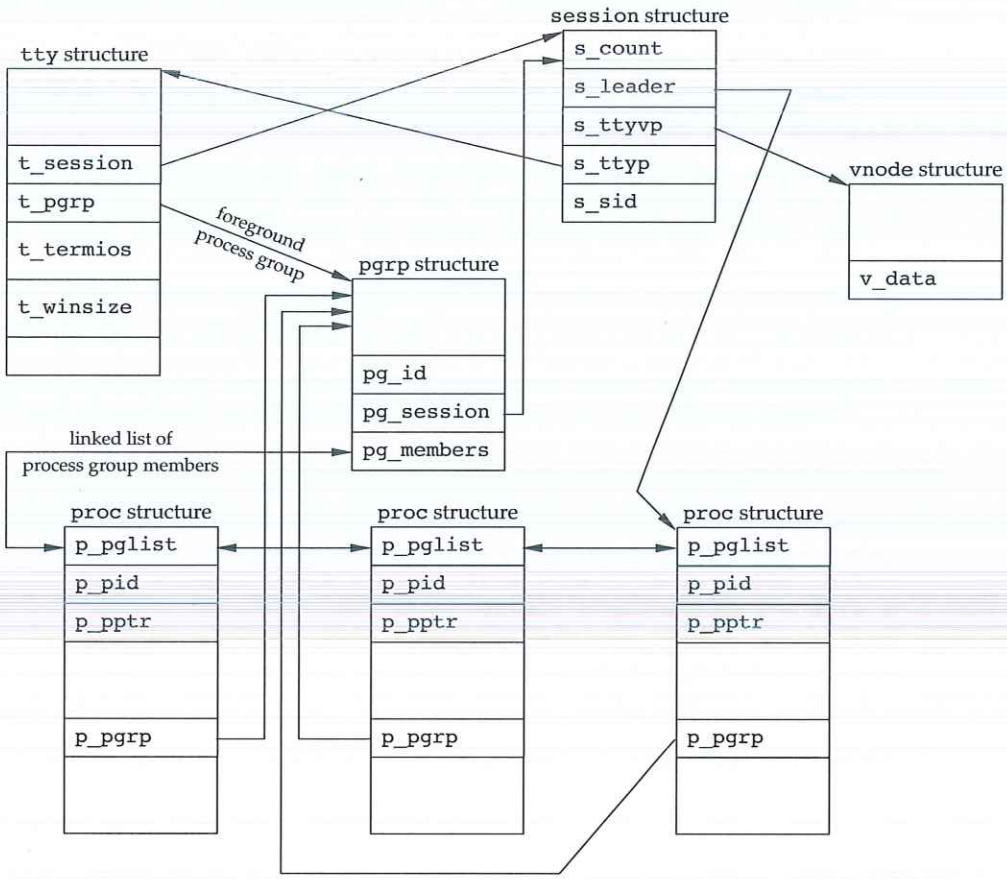

Figure 9.13 FreeBSD implementation of sessions and process groups

Let's look at all the fields that we've labeled, starting with the `session` structure. One of these structures is allocated for each session (e.g., each time `setsid` is called).

- `s_count` is the number of process groups in the session. When this counter is decremented to 0, the structure can be freed.

- `s_leader` is a pointer to the `proc` structure of the session leader.
- `s_ttyvp` is a pointer to the `vnode` structure of the controlling terminal.
- `s_ttyp` is a pointer to the `tty` structure of the controlling terminal.
- `s_sid` is the session ID. Recall that the concept of a session ID is not part of the Single UNIX Specification.

When `setsid` is called, a new `session` structure is allocated within the kernel. Now `s_count` is set to 1, `s_leader` is set to point to the `proc` structure of the calling process, `s_sid` is set to the process ID, and `s_ttyvp` and `s_ttyp` are set to null pointers, since the new session doesn't have a controlling terminal.

Let's move to the `tty` structure. The kernel contains one of these structures for each terminal device and each pseudo terminal device. (We talk more about pseudo terminals in Chapter 19.)

- `t_session` points to the `session` structure that has this terminal as its controlling terminal. (Note that the `tty` structure points to the `session` structure, and vice versa.) This pointer is used by the terminal to send a hang-up signal to the session leader if the terminal loses carrier (Figure 9.7).
- `t_pgrp` points to the `pgrp` structure of the foreground process group. This field is used by the terminal driver to send signals to the foreground process group. The three signals generated by entering special characters (interrupt, quit, and suspend) are sent to the foreground process group.
- `t_termios` is a structure containing all the special characters and related information for this terminal, such as baud rate, whether echo is enabled, and so on. We'll return to this structure in Chapter 18.
- `t_winsize` is a `winsize` structure that contains the current size of the terminal window. When the size of the terminal window changes, the `SIGWINCH` signal is sent to the foreground process group. We show how to set and fetch the terminal's current window size in Section 18.12.

To find the foreground process group of a particular session, the kernel has to start with the session structure, follow `s_ttyp` to get to the controlling terminal's `tty` structure, and then follow `t_pgrp` to get to the foreground process group's `pgrp` structure. The `pgrp` structure contains the information for a particular process group.

- `pg_id` is the process group ID.
- `pg_session` points to the `session` structure for the session to which this process group belongs.
- `pg_members` is a pointer to the list of `proc` structures that are members of this process group. The `p_pglist` structure in that `proc` structure is a doubly linked list entry that points to both the next process and the previous process in the group, and so on, until a null pointer is encountered in the `proc` structure of the last process in the group.

The `proc` structure contains all the information for a single process.

- `p_pid` contains the process ID.
- `p_pptr` is a pointer to the `proc` structure of the parent process.
- `p_pgrp` points to the `pgrp` structure of the process group to which this process belongs.
- `p_pglist` is a structure containing pointers to the next and previous processes in the process group, as we mentioned earlier.

Finally, we have the `vnode` structure. This structure is allocated when the controlling terminal device is opened. All references to `/dev/tty` in a process go through this `vnode` structure.

9.12 Summary

This chapter has described the relationships between groups of processes—sessions, which are made up of process groups. Job control is a feature supported by most UNIX systems today, and we've described how it's implemented by a shell that supports job control. The controlling terminal for a process, `/dev/tty`, is also involved in these process relationships.

We've made numerous references to the signals that are used in all these process relationships. The next chapter continues the discussion of signals, looking at all the UNIX System signals in detail.

Exercises

9.1 Refer back to our discussion of the `utmp` and `wtmp` files in Section 6.8. Why are the logout records written by the `init` process? Is this handled the same way for a network login?

9.2 Write a small program that calls `fork` and has the child create a new session. Verify that the child becomes a process group leader and that the child no longer has a controlling terminal.