Advanced Programming in the UNIX Environment — *Process Relationships*

Hop Lee
hoplee@bupt.edu.cn

Contents

1 Terminal Login

- The procedure that we describe is used to log in to a UNIX system using an terminal.
- When the system is bootstrapped the kernel creates process ID 1, the `init` process, and it is `init` that brings the system up multiuser.
- For every terminal device that allows a login in the file `/etc/ttys`, `init` does a `fork` followed by an `exec` of the program `getty`.
- All the processes shown in Figure 1 have a real user ID 0 and an effective user ID 0.
- The terminal is opened for reading and writing by `getty` calling `open` for the terminal device.
- Once the device is opened, file descriptors 0, 1, and 2 are set to the device.
- Then `getty` outputs some common information about the system (`/etc/issue`) and a string like `login:` and waits for us to enter our user name.
- `getty` is done when we enter our user name. It then invokes the `login` program, similar to

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

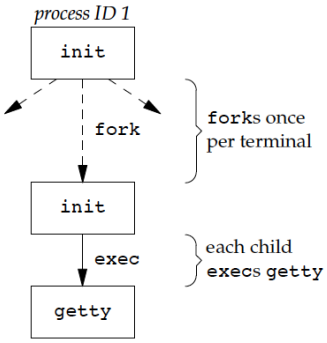

Figure 1: Process invoked by `init` to allow terminal logins.

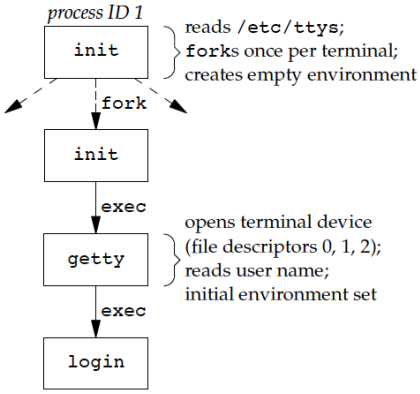

Figure 2: State of processes after `login` has been invoked

- All the processes shown in Figure 2 have superuser privileges.
- The process ID of the bottom three processes is the same.
- The `login` can call `getpwnam` to fetch our password file entry.
- Then calls `getpass` to display the prompt `Password:` and read our password.
- It calls `crypt` to encrypt the password that we entered and compares the result with the `pw_passwd` field.
- If the login attempt fails (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` again.
- If we login correctly, `login` changes to our home directory (`chdir`), the ownership of our terminal device is changed (`chown`), the access permissions are also changed for the terminal device, our group IDs are set (`setgid` and then `initgroups`), the environment is initialized.
- Finally it changes to our user ID (`setuid`) and invokes our login shell, as in

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

- `login` really does more than we've described here. It optionally prints the message-of-the-day file (`/etc/motd`), checks for new mail, and does other functions.

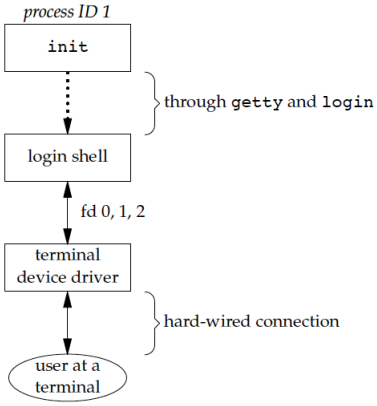

Figure 3: Arrangement of processes after everything is set for a terminal login

- Our login shell now reads its start-up files. These start-up file usually change come of the environment variables and add many additional variables to the environment.

2 Network Login

- With the terminal logins, `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, and we do not know ahead of time how many of these will occur.
- And there is'nt any terminal ready for network logins.
- To resolve the problem, `init` invokes a shell to execute a script, and then create a child process named `inetd` (or `xinetd`).

- `inetd` waits for TCP/IP connection requests. When such a request arrives the host, it does a `fork` and `exec` of the appropriate program, eg. `telnetd`.

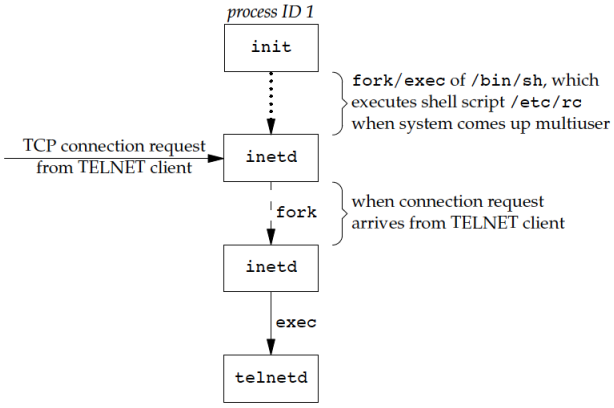

Figure 4: Sequence of processes involved in executing TELNET server

- 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.

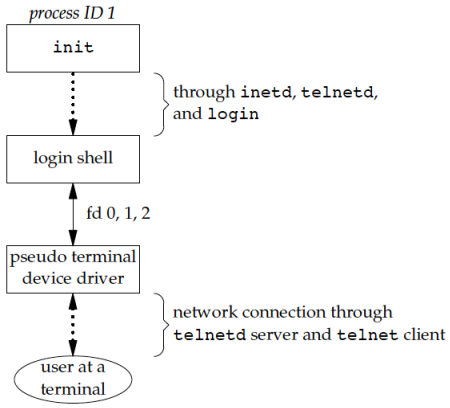

Figure 5: Arrangement of processes after everything is set for a network login

- All the procedures we just talked about is based on FreeBSD, and the other three experimental system are almost the same as it.

3 Process Groups

- In addition to having a process ID, each process also belongs to a **process group**.
- A process group is a collection of one or more processes. Each process group has a unique process group ID. The function `getpgrp` returns the process group ID of the calling process, and the `getpgrp` function took a *pid* argument and returned the process group for that process.

```
1 #include <unistd.h>
2 pid_t getpgrp(void);
3 pid_t getpgid(pid_t pid);
```

- Each process group have a **process group leader**. The leader is identified by having its process group ID equal its process ID.
- The process group still exists, as long as there is at least one process in the group, regardless whether the group leader terminates or not.
- 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 in the group either terminate or enter some other process group.
- A process joins an existing process group, or creates a new process group by calling `setpgid`

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

- This sets the process group ID to *pgid* of the process *pid*. If the two arguments are equal, the process specified by *pid* becomes a process group leader.
- A process can set the process group ID of only itself or one of its children.
- 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.

4 Sessions

- A **session** is a collection of one or more process groups.

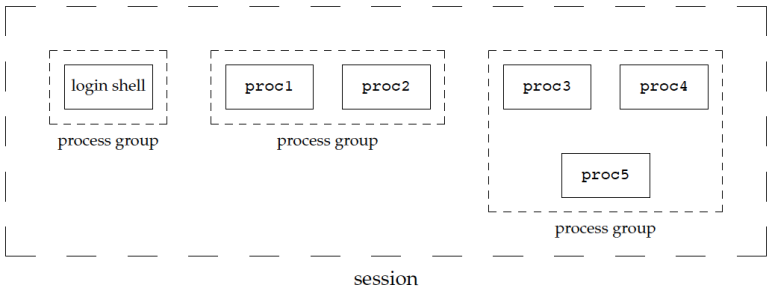

Figure 6: Arrangement of processes into process groups and sessions

- The processes in a process group are usually grouped together into the process group 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);
```

- 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.
- 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.
2. The process becomes the **process group leader** of a new process group.
3. The process has no **controlling terminal**. If the process had a controlling terminal before calling `setsid`, that association is broken.

- SUS talks about a session ID that is the process ID of the **session leader**. The `getsid` function returns the process group ID of a process's session leader.

```
1 #include <unistd.h>
2 pid_t getsid(pid_t pid);
```

- It 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.

5 Controlling Terminal

- A session can have a single **controlling terminal**. This is usually the terminal device or pseudo-terminal device 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, then it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type our terminal's interrupt key or quit key, this causes either the interrupt or quit signal to be sent to all processes in the foreground process group.
- If a modem/network disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process.

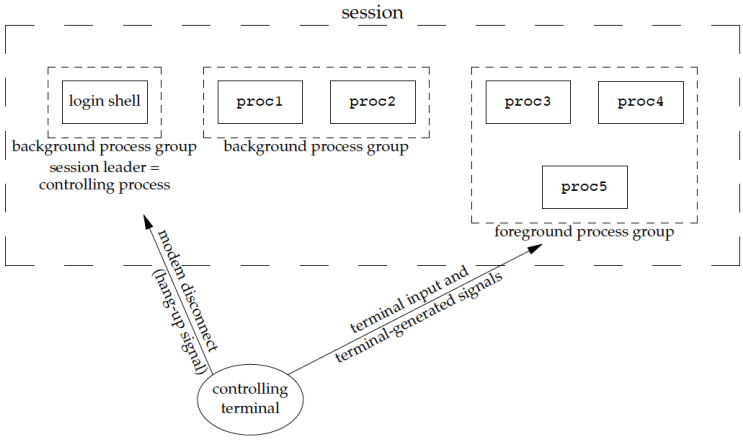

Figure 7: Arrangement of processes into process groups and sessions

- Usually, we don't have to worry about the controlling terminal; it is established automatically when we log in.
- The way a program guarantees that it is talking to the controlling terminal is to open `/dev/tty`.

6 `tcgetpgrp` and `tcsetpgrp` Functions

- The `tcgetpgrp` function can 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.

```
1 #include <unistd.h>
2 pid_t tcgetpgrp(int fd);
3 int tcsetpgrp(int fd, pid_t pgrpid);
```

- 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.

```
1 #include <termios.h>
2 pid_t tcgetsid(int fd);
```

- It 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.

7 Job Control

- Job Control means we can start multiple jobs (groups of processes) from a single terminal and 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
- Not all hosts support job control and not all shells support job control.
- The command `bg`, `fg`, `jobs`, and suspend character were prepared for job control under `bash`.
- 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.

- When we enter characters at a 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.
- When a background job sends its output to the controlling terminal, there is an option that we can allow or disallow. Normally, we use the `stty`(1) command to change this option.
- Check Figure 9.9 in textbook for summary of the features of job control.

8 Shell Execution of Programs

- A long (complicated) example on textbook, p303-p307, shows how the shells execute programs and how this relates to the concepts of process groups, controlling terminals, and sessions.

9 Orphaned Process Groups

- 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.
- 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.
- 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.
- 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` in this situation.
- The child was placed in a background process group when the parent terminated.

The End of Chapter 9.