Process Relationship Notes
From Advanced Programming in the Unix Environment 3e Chapter 9

Tyler Ryan

February 6, 2023

Contents

1 Process Relationsihps Summary

In the previous chapter 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 perent can obtain the child's exit status.

The previous chapter also mentioned process groups when it described the waitpid function and how we can wait for any process in a process group to terminate.

This chapter describes 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, isalso involved in these process relationsips.

We've mad enumerous 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.

2 Terminal Logins

2.1 BSD Terminal Logins

- System Admin creates a file, usually /etc/ttys, that has one line per temrinal devices.
  - Each line specifies the name of the devices and otehr parametes that are passed to the getty program.
  - One parameter is the "baud rate" of the terminal.
- When the system is bootstrapped, the kernel creates the Process ID 1, the init process.
  - This init process brings the system up multiuser.
  - This init process reads the file /etc/ttys
  - For every terminal deivces that allows a login, does a fork() followed by an exec() of the program getty.
  - It execs the getty program with an empty environment.
- All processes at this point have superuser privileges.
  - I.e. UID == 0 and EUID == 0
- It is getty that calls open() for the terminal devices.
  - 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 opened, the device descriptors 0, 1, and 2 are set to the devices.
- getty output something like "login:" and waits for the user to enter their username.
- When a username is entered, getty's job is complet, and it the ninvokes the "login" program similar to:
  - execle("/bin/login", "login", "-p", username, (char*) 0, envp);

continue whenever. Doesn't seem super important

2.2 Mac OS X Terminal Logins

2.3 Linux Terminal Logins

2.4 Solaris Terminal Logins

3 Process Groups

3.1 Function Prototypes

Listing 1: Process Group Function Prototypes

```
#include <unistd.h>

/* If pid == 0, the PID of the calling process is reutrned.
 * Thus getpgid(0) == getpgrp();
 */
pid_t getgrp(void);
pid_t getpgid(pid_t pid);

/* A process joins an existing process or creates a new process
 * group by calling setgpid().
 *
 * Sets the PID to pgid in the process whose PID equals "pid".
 * If the two arguments are equal, the process specified
 * by "pid" becomes a process group leader.
 */
int setpgid(pid_t pid, pid_t pgid);
```

3.2 About Process Group

- A process group is a collection of one or more processses, usually associated with the same job that can receive signals from the same terminal.
- Each process group has a unique process group ID.
- Process Groups can have a process group leader.
  - The leader is identified by its process group ID (PGID) being equal to its process ID (PID).
- It is possible for a process group leader to create a process group, create processes in the group, and then terminate them.
- The process group still exists as long as at least one process is in the group, regardless of whether the group leader terminates.
- 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 or creates a new process gropu by calling setgpid().
- A process can set the process group ID PGID of only itself or any of its children.
- A process can't change the process group ID PGID of one of its children after that child has called on of the exec() functions.

4 Sessions

4.1 Function Prototypes

Listing 2: Session Function Prototypes

```
#include <unistd.h>
    pid_t setsid(void);
        Returns: process group ID if OK, -1 on error
    pidt getsid(pid_t pid);
        Returns: session leader's process group ID PGID if OK, -1 on error.
```

- A session is a collection of one or more process groups.
- setsid() returns an error if the caller is already a Process Group Leader.
- It is impossible for a child's PID to equal its inherited PGID.
- getsid(0); returns the process group ID PGID of the calling process's session leader.
- If the calling process of setsid() is not a process group leader, this function creates a new session and the following 3 things happen:
  1. The process becomes the "session leader" of this new 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 PGID is the process ID of the calling process.
  3. The process has no controlling terminal.
     - If the process had a controlling terminal before calling setsid(), that association is broken.

5 Controlling Terminal

GET Figure 9.7 HERE

- Controlling Terminal: Usually the terminal device (in the case of aterminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- A session can have 1 Controlling Terminal.
- Controlling Process: A Session Leader that establishes the connection to the controlling terminal.
- The process groups within a session can be divided into:
  - 1 Foreground Process Group
  - $\geq$ 1 Background Process Group(s)
  - If a session has a Controlling Terminal,it has a single foreground process group, and all other process gropus in the session are background process groups.
- Whenever the terminal's interrupt key (DELETE or Control-C) is pressed, this causes the interrupt signal to be sent to all process in the FOREGROUND Process Group.
- If a modem (or network) disconnect and is detected by the terminal interface, the hang-up signal (SIGHUP) is sent to the Controlling Process (ie The Session Leader).
- Usually, we don't have to worry about the Controlling Terminal; it is established automatically when we login.
- open() the file /dev/tty guarantees that a program is talking to the Controlling Terminal.
  - Thss special file is a synonym within the kernel for the Contorlling Temrinal.
  - If the program doesn't have a Controlling Terminal, the open() of this device will fail.

6 Foreground Process Group Functions

Listing 3: Foreground Process Group Function Prototypes

```
#include <unistd.h>

// filedes == file descriptor
pid_t tcgetpgrp(int filedes);
    Returns: PGID of foreground process group if OK, -1 on error
int tcsetpgrp(int fildes, pid_t pgrpid);
    Returns: 0 if OK, -1 on error
pid_t tcgetsid(int filedes);
    Returns: Session Leader's PGID if OK, -1 on error.
```

- These functions tell the kernel which process group is in the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals.
- tcgetpgrp() returns the PGID of the foreground process group associated with the termianl open on "filedes".
- If the process has a Controlling Terminal, the process can call tcsetpgrp() to set the foreground PGID to pgrpid.
  - The value of pgrpid must be the process group ID PGID of a process group in the SAME SESSION, and filedes must refer to the Controlling Temrinal of the session.
- Most applications don't call these two functions directly.
  - They are normally called by Job-Control Shells.
- Applications that need to manage Controlling Terminals can use tcgetsid() to identify the Session ID of the Controlling Terminal's Session Leader
  - Controlling Terminal's Session Leader == Session Leader's PGID.

7 Job Control

- Job: Groups of Processes.
- Job Control: Allows the starting of multiple Jobs from a single terminal and to control which jobs can access the temrinal and which jobs are to run in the background.
- Job Control requires 3 forms of support:
  - A SHELL that supports Job Control
  - The TERMINAL DRIVER in the kernel must support job control
  - The KERNEL must support certain job control signals.
- 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 output while we were entering an input line.
- Terminal Driver looks for 3 special characters, which generate signals to the FOREGROUND Process Group (ONLY)
  1. DELETE or Control C: Generates SIGINT
  2. Control- Generates SIGQUIT
  3. Control-Z: Generates SIGTSTP.
- Only FOREGROUND jobs receive temrinal 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 normally stops the background process
  - Notifies the shell (i.e. us) an we can bring it to the foreground.

7.1 Job Control Examples

Listing 4: Job Control Examples

```
# Foreground Process
$ ./a.out

# Background Process
$ ./a.out &

# Switching between background and foreground processes
$ cat > temp.txt &              # Start in the background
[1]   1234
$                               # Press Return
[1] + Stopped    cat > temp.foo
$ fg 1                          # Move to the foreground
hello, world                    # Type in "hello, world"
^D                              # End of File (EOF) character
$ cat temp.txt                  # Observe the output
hello,world
```

- Starts **cat** process in the background,
- But when **cat** tries to read its standard input (the Contolling Terminal), the Terminal Driver, knowing that it is a background job, sends the SIGTTIN signal to the background job.
- The shell detects the change in status of its child (**wait()** and **waitpid()**) and tells us that the job has been stopped.
- Move the job into the foreground with the shell's **fg** command.
  - Doing so uses the **tcsetpgrp()** function to place the job into the foreground process group.
- The shell sends the signal SIGCONT to the process group.
- Since the job is now in the foreground process group, it can read from the Controlling Terminal.

Note: If the background job outputs to the controlling terminal, you can disable it with the **stty tostop** command.

8 Shell Excecution Of Programs

8.1 Example with the Bourne Shell

Command:

```
$ ps -o pid,ppid,pgrp,session,tpgid,comm | cat1
        # Look at their PPIDs and PIDs.
        # This shows that "ps" and "cat1" are children of "cat2"
        # and that "cat2" is a child of the shell.
```

PID	PPID	PGID	SID	COMMAND
2837	2818	2837	2837	5799
5799	2837	**5799**	2837	5799
5800	2837	**5799**	2837	5799

8.2 Foreground Example With The Bourne Again Shell (BASH)

Command:

```
$ ps -o pid,ppid,pgrp,session,tpgid,comm | cat1
```

PID	PPID	PGRP	SESS	TPGID	COMMAND
2837	2818	2837	2837	5799	bash
5799	2837	**5799**	2837	5799	ps
5800	2837	**5799**	2837	5799	cat1

- This example starts the jobs in the foreground.
- Foreground processes are in bold.
- Both processes, "ps" and "cat1", are placed into the new process group (PGRP) 5799 (ie. the Foreground group)
- "ps" was also created first instead of what would've been last if cat2 wasn't there for the Bourne Shell.

8.3 Background Example With The Bourne Again Shell (BASH)

This example starts the jobs in the background.
Command:

```
$ ps -o pid,ppid,pgrp,session,tpgid,comm | cat1 &
```

PID	PPID	PGRP	SESS	TPGID	COMMAND
2837	2818	**2837**	2837	2837	bash
5801	2837	5801	2837	2837	ps
5802	2837	5801	2837	2837	cat1

- This example starts the jobs in the background.
- Foreground processes are in bold.
- Both processes, "ps" and "cat1", are placed into the BACKGROUND.
- "ps" was also created first instead of what would've been last if cat2 wasn't there for the Bourne Shell.

9 Orphaned Process Groups

- A PROCESS whose parent terminates is called an orphan and is inherited by the init process.
- 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.
  - In other words, 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.
- POSIX.1 requires that every process in the newly orphaned process group that is stopped to be sent the hang-up signal (SIGHUP) followed by the continue signal (SIGCONT).

10 Example Code

Listing 5: Orphaned Process Example

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

/* This program creates an orphaned process by ending
 * a parent process while it's child is stopped.
 */

// What to do if SIGHUP is sent
static void sig_hup(int signo)
{
    printf("SIGHUP received, pid = %d\n", getpid());
}

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

int main()
{
    char  c;
    pid_t pid;

    // First to print
    pr_ids("parent");

    pid = fork();

    if (pid < 0)
    {
        perror("Error");
        exit(EXIT_FAILURE);
    }

    // Parent
    else if (pid > 0)
    {
        sleep(2);
        printf("Parent process ended\n");
```

```
        exit(EXIT_SUCCESS);
    }

    // Child
    else
    {

        // Second to print
        pr_ids("child");

        /* POSIX.1 requires that every process in the newly orphaned
         * process group that is STOPPED (as our child is) be sent
         * the hang-up signal (SIGHUP) FOLLOWED BY THE CONTINUE
         * SIGNAL (SIGCONT).
         *
         * The default action of SIGHUP is to terminate the process.
         * This would terminate our program.
         *
         * signal() tells the program what to do in with a
         * particular signal. Essentially allowing us to override
         * the default action of any given signal.
         *
         * To override the default action of SIGHUP, we pass our
         * sig_hup() function, which does not terminate the process,
         * to signal(). Thus allowing it to continue.
         *
         * NOTE: Without signal() or kill(), the child would still
         * continue in the background even after the parent ended.
         *
         * In short, the kill() function is being ran and thus
         * our child is stopped. However, it becomes an orphan
         * after the parent process ends. Every stopped orphan
         * must be continued. Thus is why our child continues.
         */

        // signal() looks to be a higher order function.
        signal(SIGHUP, sig_hup);
        printf("After signal()\n");

        printf("Program Stopped\n");
        kill(getpid(), SIGTSTP);
        printf("Continue after kill() (Being Stopped)\n");


        /* I believe this is trying to illustrate the fact that because
         * the this process is orphaned, it is sent to the background.
         * However, a background process cannot take input from the
         * controlling terminal so it errors.
```

```
         *
         * To get this to work, you need to enter a single letter
         * before the parent exits.
         */

        /* ssize_t status = read(STDIN_FILENO, &c, 1); */

        /* if (status != 1) */
        /*     printf("read error from controlling TTY, errno = %d\n",
                                                    *     errno); */


        /* This demonstrates how the process is run in the background
         * even after the parent ends.
         */

        printf("Continuing in the background.\n");
        sleep(3);
        pr_ids("\nchild");
        printf("Child process ends\n");
        printf(
        "Notice how the Parent PID (PPID) changes to 1 (ie. init)\n"
        );
        exit(EXIT_SUCCESS);
    }
}
```

Output:

```
parent: pid = 366545, ppid = 366503, pgrp = 366545, tpgrp = 366545
child: pid = 366546, ppid = 366545, pgrp = 366545, tpgrp = 366545
After signal()
Program Stopped
Parent process ended
SIGHUP received, pid = 366546
Continue after kill() (Being Stopped)
Continuing in the background.

$
child: pid = 366546, ppid = 1, pgrp = 366545, tpgrp = 366503
Child process ends
Notice how the Parent PID (PPID) changes to 1 (ie. init)
```

11 FreeBSD Implementation

11.1 Nothing Here

12 Exercises

12.1 Exercise 1

1. Refer back to our discussion of the utmp and wtmp files. Why are the logout records written by the init process?
Is this handled the same way for a newtwork login?

12.2 Exercise 2

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.