MODULE 4

Changing User IDs and Group IDs, Interpreter Files, system Function, Process Accounting, User Identification, Process Times, I/O Redirection.

CHANGING USER IDs AND GROUP IDs

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

```
#include <unistd.h>
```

int setuid(uid_t uid); int setgid(gid_t gid); Both return: 0 if OK, 1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user- ID, setuidsets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.
- If neither of these two conditions is true, errnois set to EPERM, and 1 is returned.

We can make a few statements about the three user IDs that the kernel maintains.

Only a superuser process can change the real user ID. Normally, the real user ID is set by the login(1) program when we log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid.

The effective user ID is set by the execfunctions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the exec functions leave the effective user ID as its current value. We can call setuidat any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally,
we can't set the effective user ID to any random value.

The saved set-user-ID is copied from the effective user ID by exec. If the file's set-user-ID bit is set, this copy is saved after execstores the effective user ID from the file's user ID.

				user
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program	set to uid	set to uid

saved set-user ID	copied from effective user ID	copied from effective user ID	set to uid	unchanged

The above figure summarises the various ways these three user IDs can be changed

setreuidand setregidFunctions

Swapping of the real user ID and the effective user ID with the setreuid function.

#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);

Both return : 0 if OK, -1 on error

We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged. The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations.

seteuid and setegid functions :

POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

#include <unistd.h>

int seteuid(uid_t uid); int setegid(gid_t gid); Both return : 0 if OK, 1 on error

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged

user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three user IDs.)

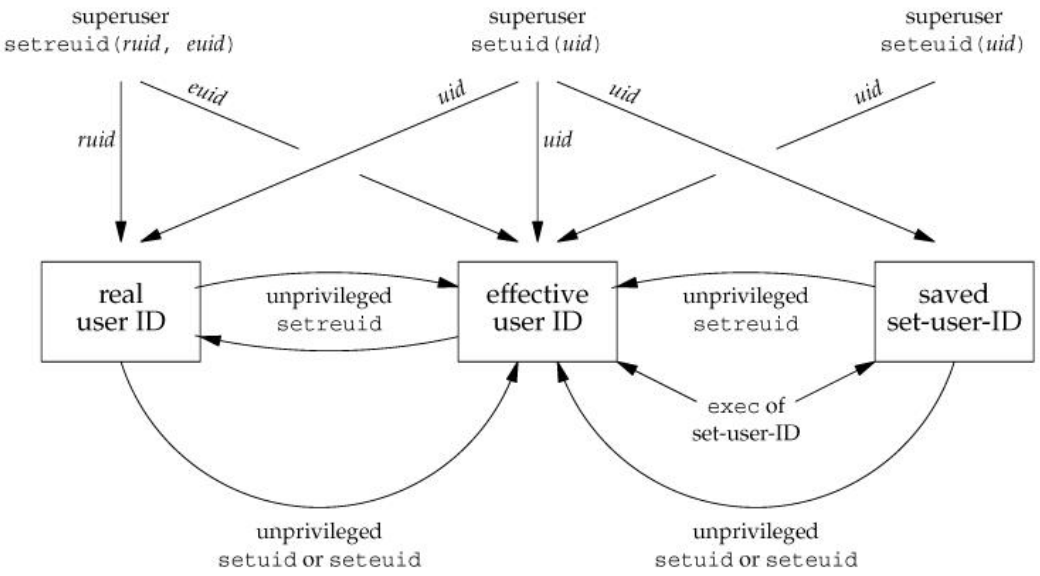


Figure: Summary of all the functions that set the various user Ids

INTERPRETER FILES

These files are text files that begin with a line of the form

#! pathname [optional-argument]

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

#!/bin/sh

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATHis not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file

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

Int main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {        /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Output:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg argv[1]: foo
argv[2]: /home/sar/bin/testinterp argv[3]: myarg1
argv[4]: MY ARG2
```

systemFUNCTION

```
#include <stdlib.h>

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

If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, systemis always available.

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

- If either the forkfails or waitpidreturns an error other than EINTR, systemreturns 1 with errnoset to indicate the error.
- If the execfails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
- Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from systemis the termination status of the shell, in the format specified for waitpid.

Program: The systemfunction implementation using fork, exec and waitpid

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

void main()
{
       int status;
       char cmd[50];
      pid_t pid=fork();

     if (pid==0)
     {
        printf("Input the cmd\n");
        scanf("%s",cmd);
        execl("/bin/sh","sh","-c",cmd,(char *)0);
    }
    else
        waitpid(pid,&status,0);
}
```

Here **execl("/bin/sh","sh","-c", cmd, (char *) 0)**

The first parameter is the path of the child process, 2nd parameter is the name given to path

-c Read commands from the command_string (cmd) operand instead of from the standard input.

Output

```
cse@ubuntu:~$ ./a.out
Input the cmd
date
Tue Oct 16 21:35:33 PDT 2018
```

```
cse@ubuntu:~$ ./a.out
Input the cmd
who
cse     tty7        2018-10-16 21:28 (:0)
cse     pts/0       2018-10-16 21:28 (:0.0)
cse     pts/1       2018-10-16 21:34 (:0.0)

cse@ubuntu:~$ ./a.out
Input the cmd
pwd
/home/cse
```

PROCESS ACCOUNTING

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.

These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.

A superuser executes accton with a pathname argument to enable accounting.

The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.

The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork. Each accounting record is written when the process terminates. This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started. The accounting records correspond to processes, not programs.

A new record is initialized by the kernel for the child after a fork, not when a new program is executed. The structure of the accounting records is defined in the header <sys/acct.h>and looks something like

```
typedef   u_short comp_t;   /* 3-bit base 8 exponent; 13-bit fraction */

struct   acct
{
   char   ac_flag;    /* flag  */
   char   ac_stat;    /* termination status (signal & core flag only) */
    uid_t  ac_uid;     /* real user ID */
   gid_t  ac_gid;     /* real group ID */
   dev_t  ac_tty;     /* controlling terminal */
   time_t ac_btime;   /* starting calendar time */
   comp_t ac_utime;   /* user CPU time (clock ticks) */
```

```
    comp_t ac_stime;    /* system CPU time (clock ticks) */
     comp_t ac_etime;    /* elapsed time (clock ticks) */
    comp_t ac_mem;      /* average memory usage */
    comp_t ac_io;       /* bytes transferred (by read and write)  "blocks" on BSD systems */
    comp_t ac_rw;       /* blocks read or written */
                            /* (not present on BSD systems) */
    char   ac_comm[8];  /* command name: [8] for Solaris, */
                            /* [10] for Mac OS X, [16] for FreeBSD, and */
                            /* [17] for Linux */
};
```

Values for ac_flagfrom accounting record

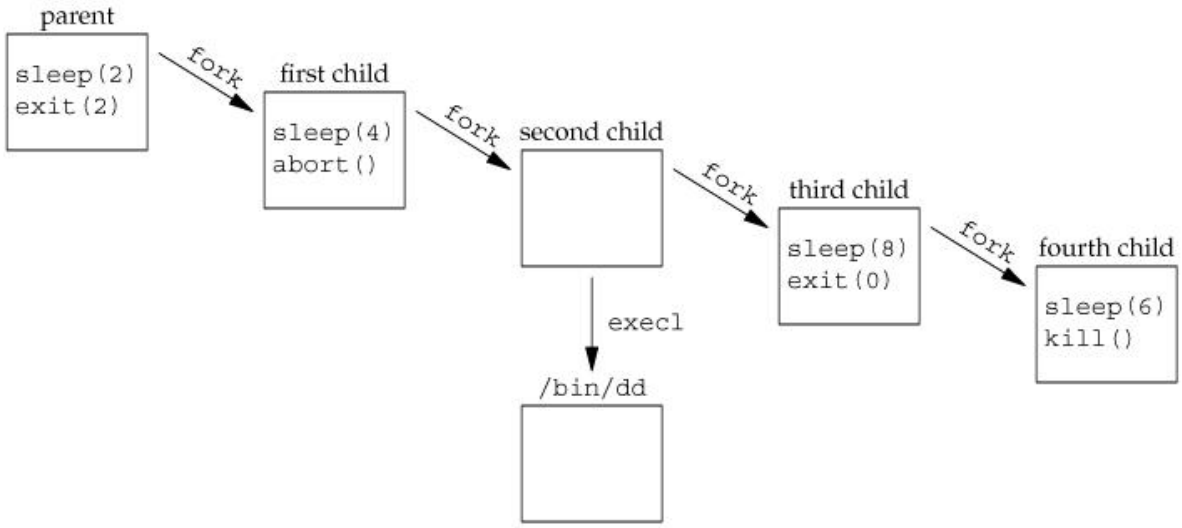
ac_flag	Description
AFORK	process is the result of fork, but never called exec
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

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

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {      /* parent */
        sleep(2);
        exit(2);            /* terminate with exit status 2 */
    }


    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
   sleep(4/* first child */
        abort();            /* terminate with core dump */
    }

    if ((pid = fork()) < 0)
```

```
            err_sys("fork error");
        else if (pid != 0) {
/* second child */
            execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
            exit(7);            /* shouldn't get here */
        }

        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid != 0) {
            sleep(8);
/* third child */
            exit(0);            /* normal exit */
        }

        sleep(6);
```

```
    kill(getpid(), SIGKILL);    /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```



Process structure for accounting example

USER IDENTIFICATION

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call getpwuid(getuid()), but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in and the getlogin function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);
```

Returns : pointer to string giving login name if OK, NULL on error

This function can fail if the process is not attached to a terminal that a user logged in to.

PROCESS TIMES

We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process

can call the times function to obtain these values for itself and any terminated

children.

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

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, 1 on error

This function fills in the tmsstructure pointed to by buf:

```
struct tms {
  clock_t   tms_utime;  /* user CPU time */
  clock_t   tms_stime;  /* system CPU time */
  clock_t   tms_cutime; /* user CPU time, terminated children */
  clock_t   tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.

PROCESS RELATIONSHIPS

INTRODUCTION

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.

TERMINAL LOGINS

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.

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 multiuser. The initprocess reads the file /etc/ttysand, 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.

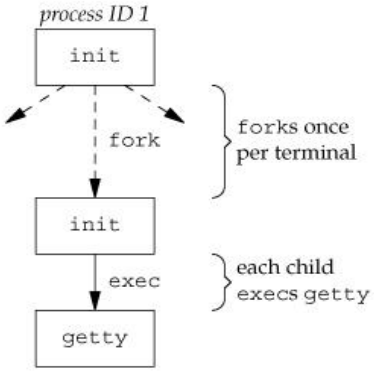


Figure 9.1. Processes invoked by init to allow terminal logins

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 initprocess also execs the gettyprogram with an empty environment. **It is gettythat calls openfor the terminal device**. The terminal is opened for reading and writing. If the device is a modem, the openmay 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 gettyoutputs something like login: and waits for us to enter our user name. When we enter our user name, getty's job is complete, and it then invokes the loginprogram, similar to

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

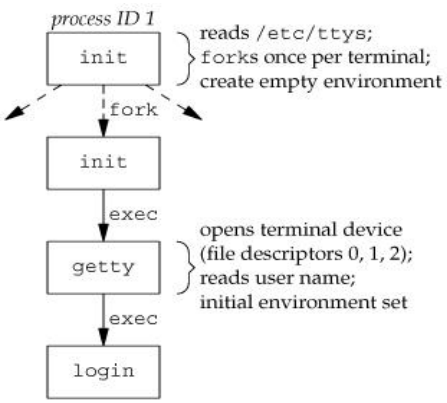


Figure 9.2. State of processes after loginhas been invoked

All the processes shown in Figure 9.2 have superuser privileges, since the original initprocess has superuser privileges.

If we log in correctly, loginwill

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 setgidand 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 they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

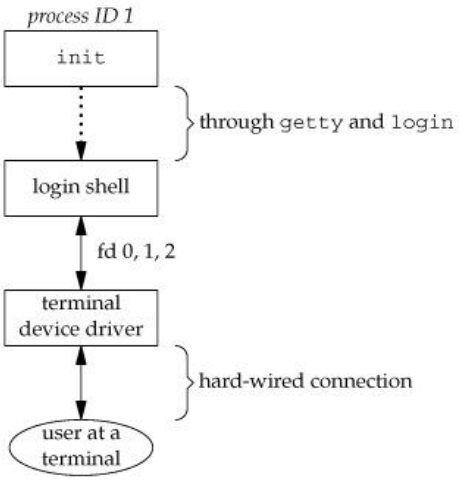


Figure 9.3. Arrangement of processes after everything is set for a terminal login

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. With the terminal logins that we described in the previous section, initknows 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).

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.

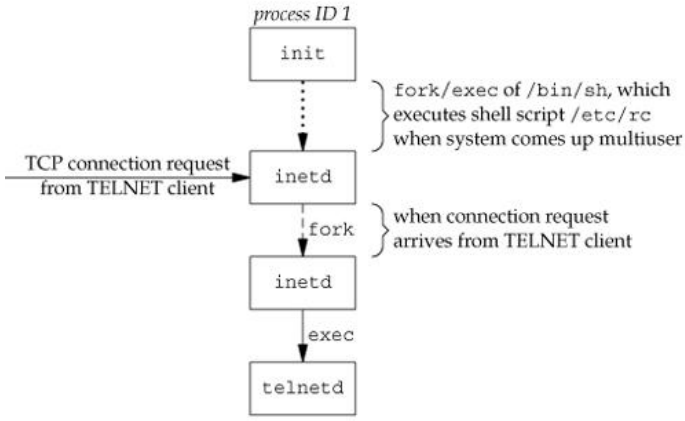
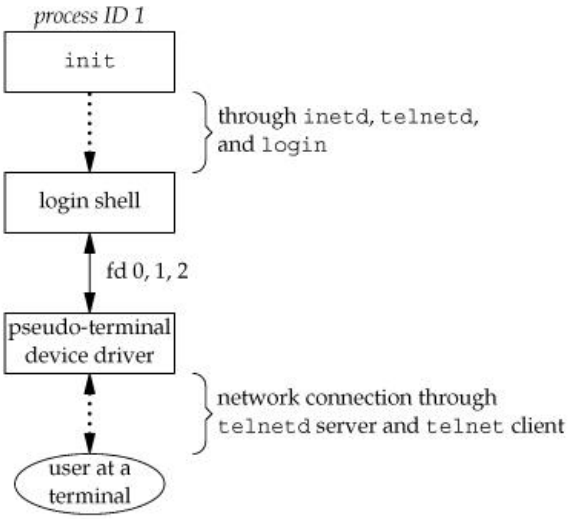


Figure 9.4. Sequence of processes involved in executing TELNET server

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, loginperforms 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



PROCESS GROUPS

A process group is a collection of one or more processes, usually associated with the same job, 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_tdata 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

The Single UNIX Specification defines the getpgid function as an XSI extension that mimics this behavior.

#include <unistd.h>

pid_t getpgid(pid_t pid);

Returns: process group ID if OK, 1 on error

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.

#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);

Returns: 0 if OK, 1 on error

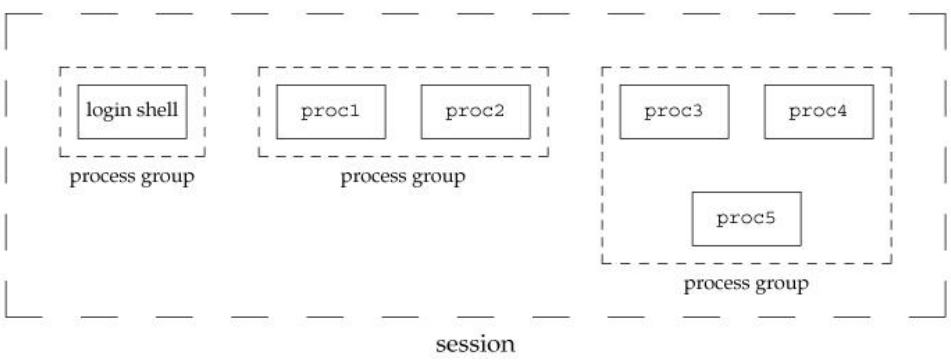
This function sets the process group ID to pgid in the process whose process ID equals pid. If the two arguments are equal, the process specified by pid becomes a process group leader. If pid is 0, the process ID of the caller is used.

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



A process establishes a new session by calling the setsidfunction.

#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. 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. 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. The process has no controlling terminal. 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. The getsid function returns the process group ID of a process's session leader. The getsidfunction is included as an XSI extension in the Single UNIX Specification.

#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, getsidreturns the process group ID of the calling process's session leader.

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.

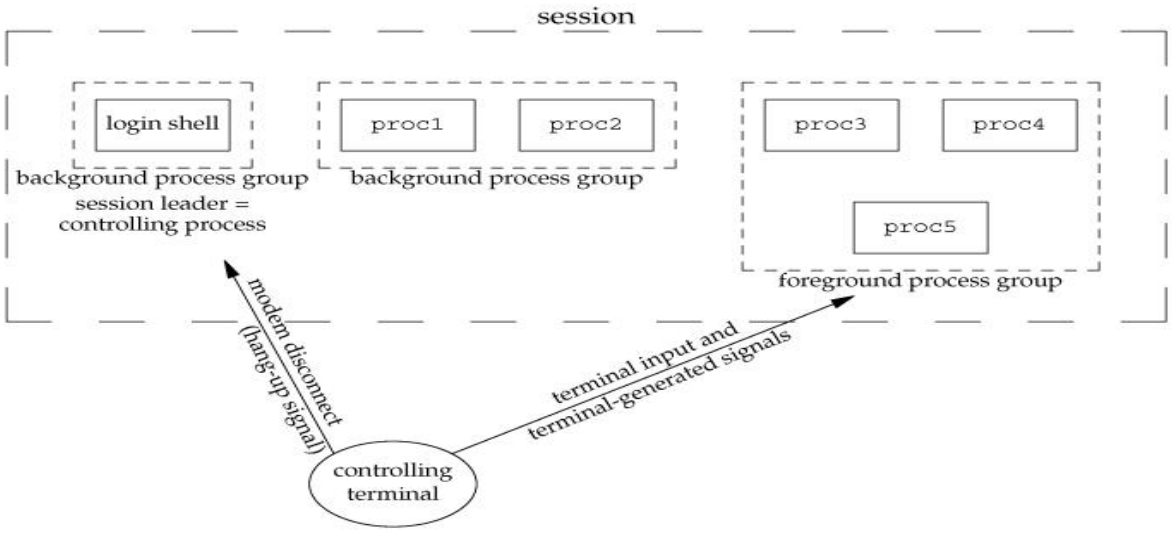
Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.

Whenever we type the terminal's quit key (often Control-backslash), this causes the

quit signal to be 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



tcgetpgrp, tcsetpgrp, AND tcgetsidFUNCTIONS

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.

To retrieve the foreground process group-id and to set the foreground process group-id we can use tcgetprgp and tcsetpgrp function.

The prototype of these functions are :

#include <unistd.h>

pid_t tcgetpgrp(int filedes);

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

int tcsetpgrp(int filedes, 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 *filedes.* 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 filedes must refer to the controlling terminal of the session.

The single UNIX specification defines an XSI extension called tcgetsid to allow an application to obtain the process group-ID for the session leader given a file descriptor for the controlling terminal.

#include <termios.h>

pid_t tcgetsid(int filedes);

Returns: session leader's process group ID if Ok, -1 on error

JOB CONTROL

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 to run in the background. Job control requires three 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 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.

This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal. The following 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 &***

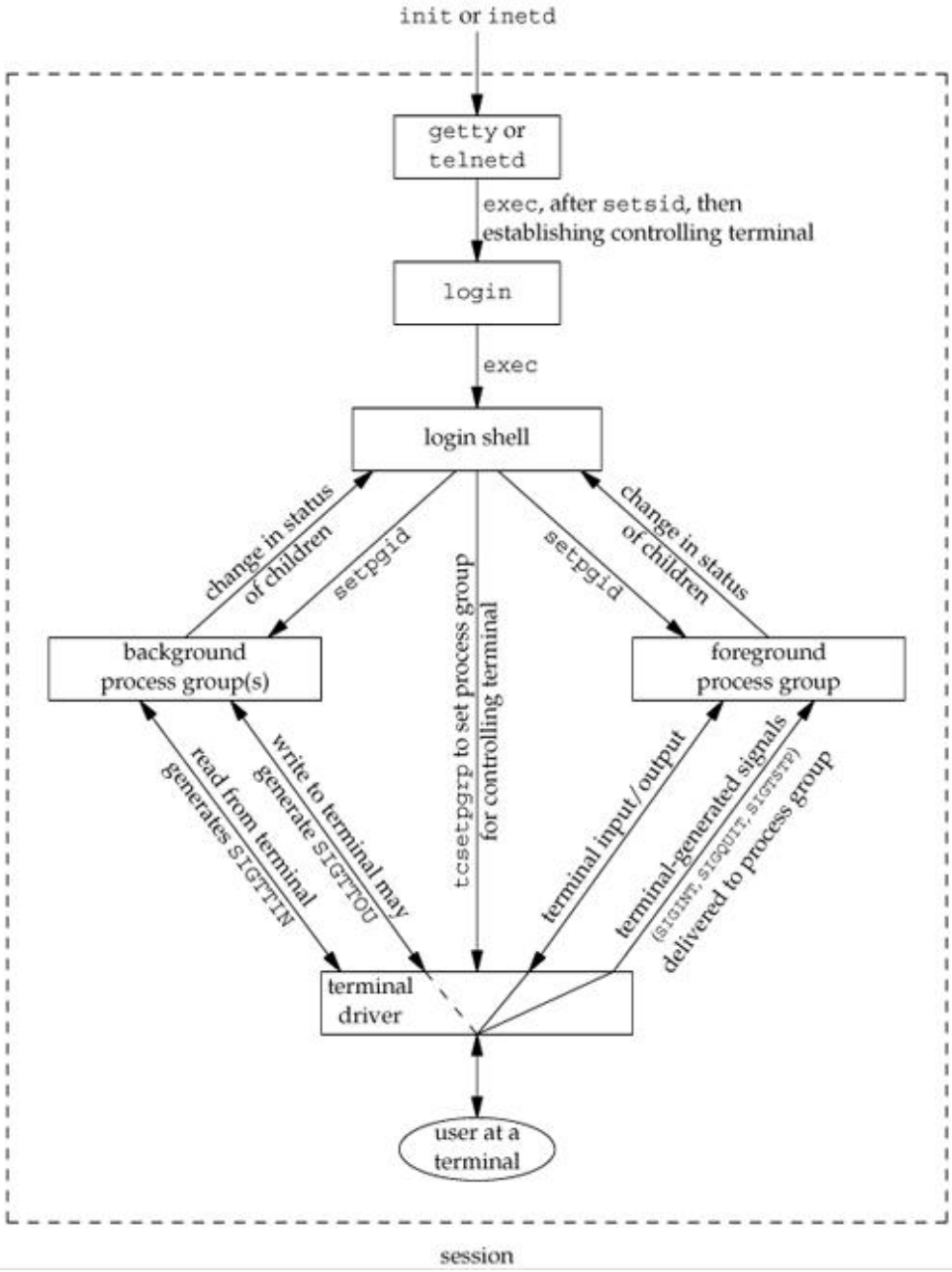
$ fg %1 ***bring job number 1 into the foreground***

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

Figure 9.8. Summary of job control features with foreground & background jobs & terminal driver



What happens if a background job outputs 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. The following 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, catwill 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 SIGTTOUsignal.

Figure 9.8 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 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.

SHELL EXECUTION OF PROGRAMS

Example 1: ps -o pid,ppid,pgid,sid,comm

Output 1:

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

Example 2: ps -o pid,ppid,pgid,sid,comm &

Output 2:

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

If we execute the command in the background, the only value that changes is the process ID of the command.

Example 3: ps -o pid,ppid,pgid,sid,comm | cat1

Output 3:

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

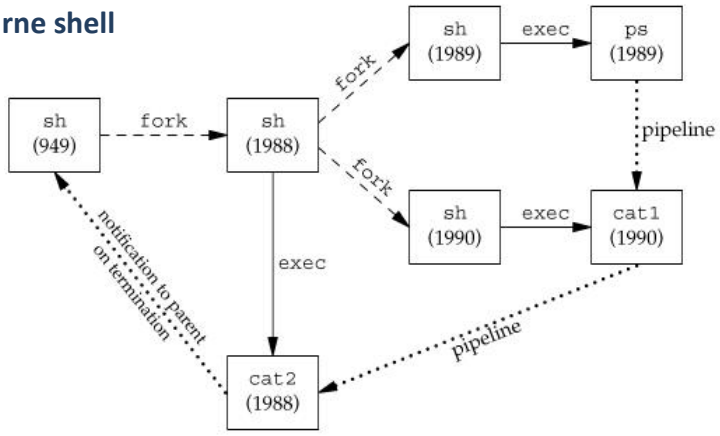
The program `cat1` is just a copy of the standard `cat` program, with a different name. Note that the last process in the pipeline is the child of the shell and that the first process in the pipeline is a child of the last process.

Example 4: ps -o pid,ppid,pgid,sid,comm | cat1 | cat2

Output 4:

```
  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
```

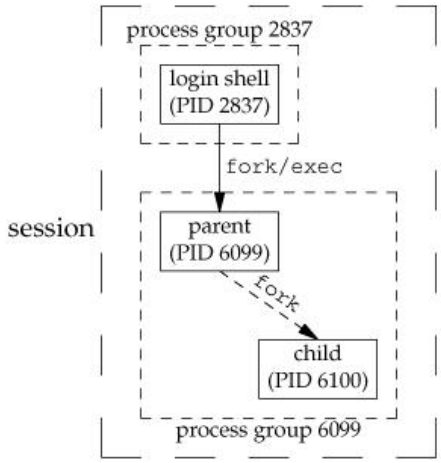
Figure 9.9. Processes in the pipeline ps | cat1 | cat2 when invoked by Bourne shell



ORPHANED PROCESS GROUPS

A process whose parent terminates is called an orphan and is inherited by the init process.

Example of a process group about to be orphaned



Creating an orphaned process group

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

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

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(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 */
            exit(0);       /* then parent exits */
        } 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 from controlling TTY, errno = %d\n", errno);
            exit(0);
        }
}
```

Output:

```
$ ./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 from controlling
 TTY, errno = 5
```

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). This is so we can see whether SIGHUP is sent to the child. 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, the init process ID.

At this point, the child is now a member of an orphaned process group 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 belongs to another session.

Since the process group is orphaned when the parent terminates, 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). 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 printfin the sig_hupfunction to appear before the printf in the pr_idsfunction.