Process Relationship

Advanced Programming in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Outline

Logins

Process groups

Sessions

Controlling terminal

Job control

Shell execution of programs

Orphaned process groups

Linux Boot Process

The first process after system boot /sbin/init

- The parent of all processes
- Has a PID of 1

/sbin/init configurations

- /etc/inittab, /etc/event.d/*, or /etc/init/*

Run levels

- sysinit
- 0 (halted), 6 (reboot), 1-5 (can be customized)
  - Default run levels – often set to 2, 3, or 5

Enable console logins

Linux Terminal Logins

Terminal setups, an example from Ubuntu 14

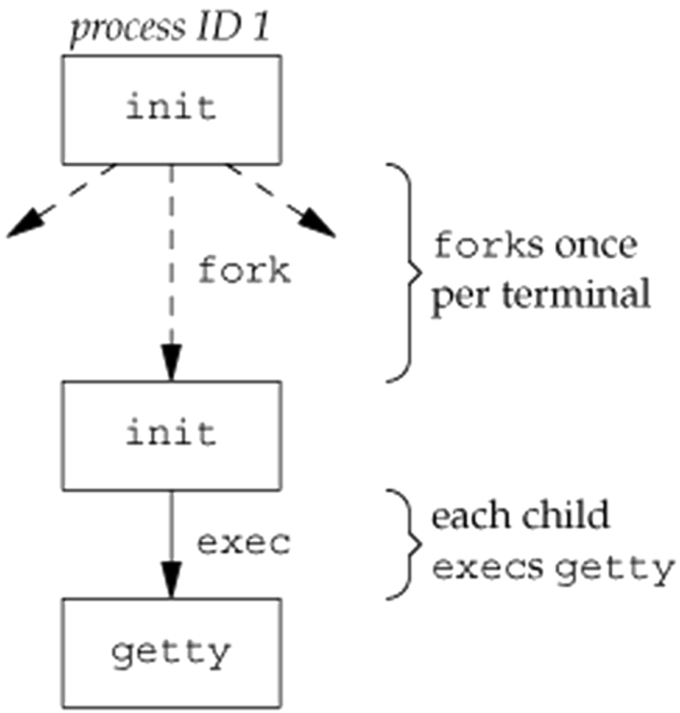
- Start 6 consoles terminals for login
- Can be switched using hotkey Alt+F1 ~ F6

/etc/event.d/tty1

```
start on stopped rc RUNLEVEL=[2345] and (
            not-container or
            container CONTAINER=lxc or
            container CONTAINER=lxc-libvirt)

stop on runlevel [!2345]

respawn
exec /sbin/getty -8 38400 tty1
```



The getty Program

Calls open for the terminal device

- /dev/tty1, /dev/ttyS0, …

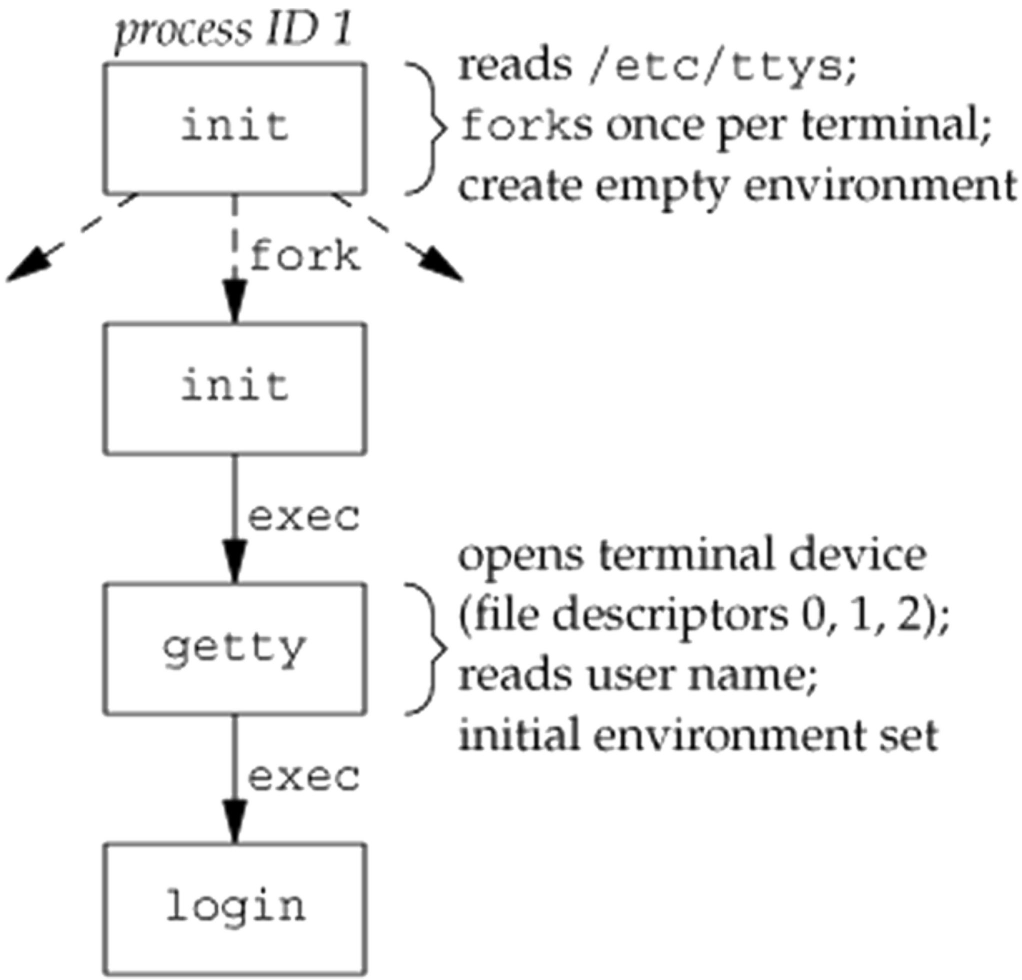
Create file descriptors 0, 1, and 2

Show the "login:" prompt

When a user provides his/her username, invoke the "/bin/login" program

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

The getty Program (Cont'd)



The login Program

Display the "Password:" prompt

- Read user password using getpass(3)
- Read encrypted password, e.g., from /etc/shadow
- Encrypt user input password, and compare the encrypted with that stored in /etc/shadow

If a user login fails …

- The login program terminates and the init restarts getty

If a user login succeedes …

- There are a lot tasks to be performed

Actions for a Successful Login

Set CWD to the user's home directory (chdir)

Set the ownership of the user's terminal device (chown)

Set the access permissions for the terminal device so the user have permission to read from and write to it

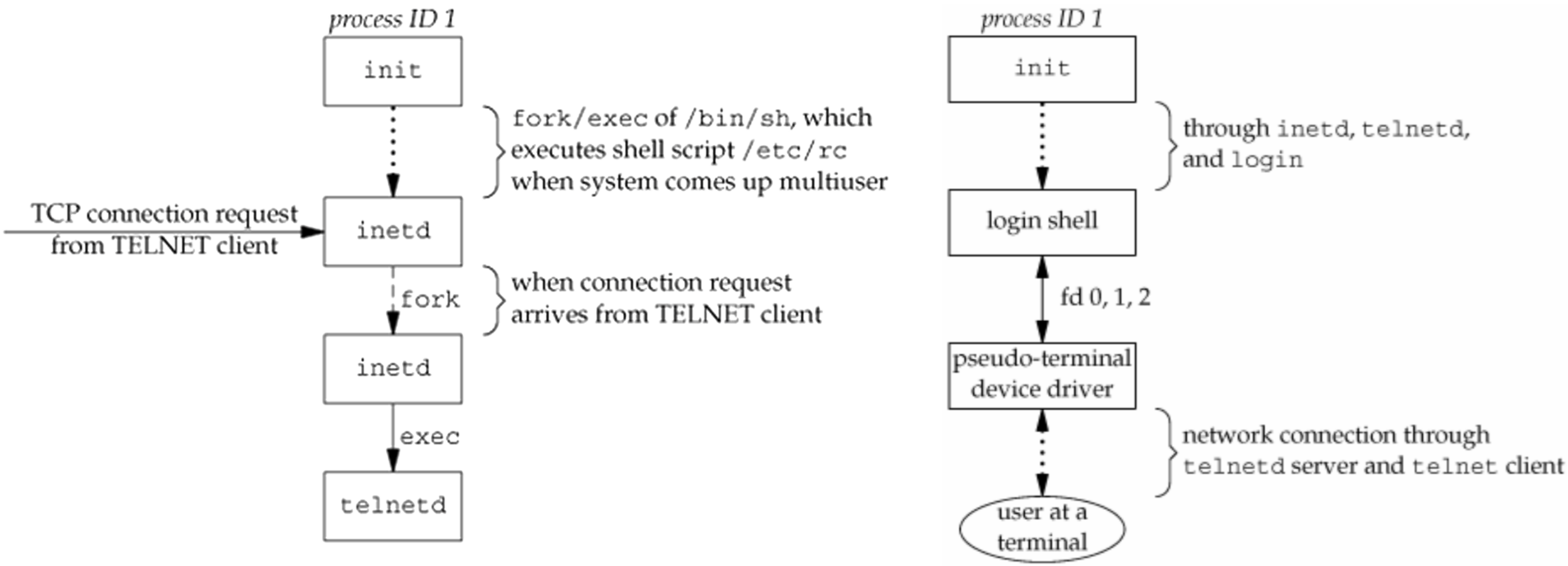
Set group IDs by calling setgid (real group) and initgroups (for supplementary groups)

Initialize the environment variables

- HOME, SHELL, USER, LOGNAME, PATH, …

Set user ID (setuid) and invoke a login shell

Network Logins – via the telnetd Program



The telnetd Program

Opens a pseudo-terminal device

- /dev/pts/N

Splits into two processes using fork

The parent handles the communication across the network connection

The child does an exec of the login program – it is the same as terminal logins

Whether we log in through a terminal or a network connection …

- We have a login shell
- Its standard input/output/error are connected to either a terminal device or a pseudo-terminal device

The Purpose of Process Group

Every process has a parent process

The parent is notified when its child terminates

The parent can obtain the child's exit status

- The waitpid function
- In addition to wait a single child, the parent process can wait children in a *process group*
- Signals (covered in the next chapter) can be also sent to processes in a process group

So, what is a process group?

What is a Process Group

Each process belongs to a process group

A process group is a collection of one or more processes
- Usually associated with the same job

Each process group has a unique process group ID

Process group IDs are similar to process IDs
- They can be stored in a pid_t data type

Retrieve of the process group ID
- `#include <unistd.h>`
- `pid_t getpgid(pid_t pid);`
- `pid_t getpgrp(void);`
  - Is equivalent to `getpgid(0);`

What is a Process Group (Cont'd)

Each process group *can* have a process group leader

- The leader is identified by its process group ID being equal to its process ID
- A group leader can create a group, create processes in the group, and then quit
- The process group still exists, as long as at least one process is in the group

The process group lifetime

- Start on the creation of the group
- End when the last process in the group leaves

Create/Join a Process Group

Synopsis

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

Explanations

- Sets the process group ID to pgid in the process whose process ID equals pid
- If pid = pgid, the process specified by pid becomes a process group leader
- If pid is 0, the process ID of the caller is used
- If pgid is 0, pgid = pid

Create/Join a Process Group (Cont'd)

setpgid Limitations

- A process can set the process group ID of only itself and any of its children
- Furthermore, it can not change the process group ID of one of its children after that child has called one of the exec functions.

The use of setpgid function

- It is called after a fork to have the parent set the process group ID of the child, and
- Have the child set its own process group ID
- The above two actions are redundant, but they guaranteed that the child is placed into its own process group

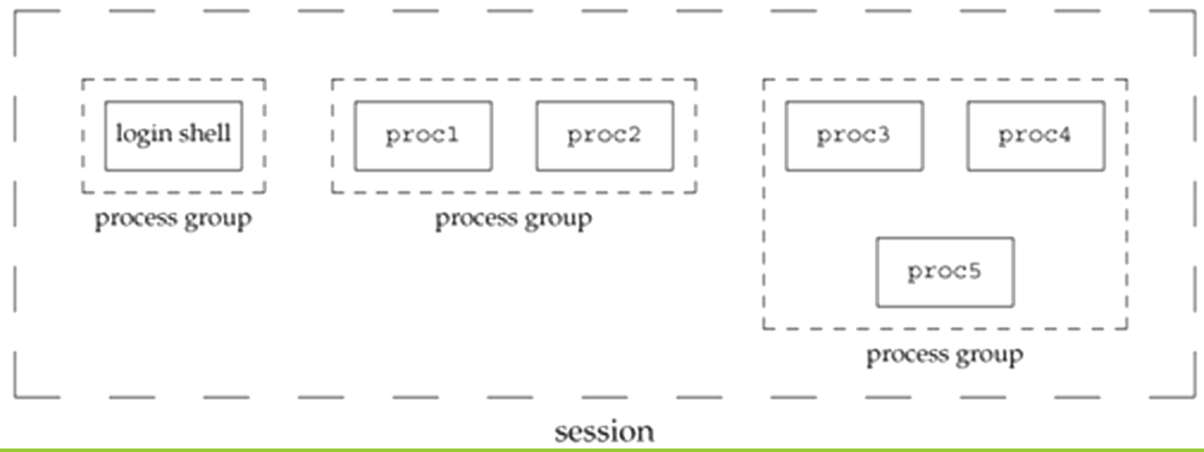
Sessions

A session is a collection of one or more process groups

The processes in a process group are usually placed there by a shell pipeline

An example

- $ proc1 | proc2 &
- $ proc3 | proc4 | proc5 &



Create a Session

Synopsis

- pid_t setsid(void);
- Returns pgid or -1 if the caller is already a process group leader

If the calling process is not a process group leader, this function creates a new session

- The process becomes the session leader of this new 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*

Get the Current Session ID

Synopsis

- pid_t getsid(pid_t pid);
- Returns the session leader’s process group ID, or -1 on error
- If pid is 0, getsid returns the process group ID of the calling process's session leader

The session ID is the process ID of the session leader

When a user logged in, the session leader is usually the shell

Controlling Terminal (1/3)

A session can have a single controlling terminal
- It is usually a terminal device or a pseudo-terminal device

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

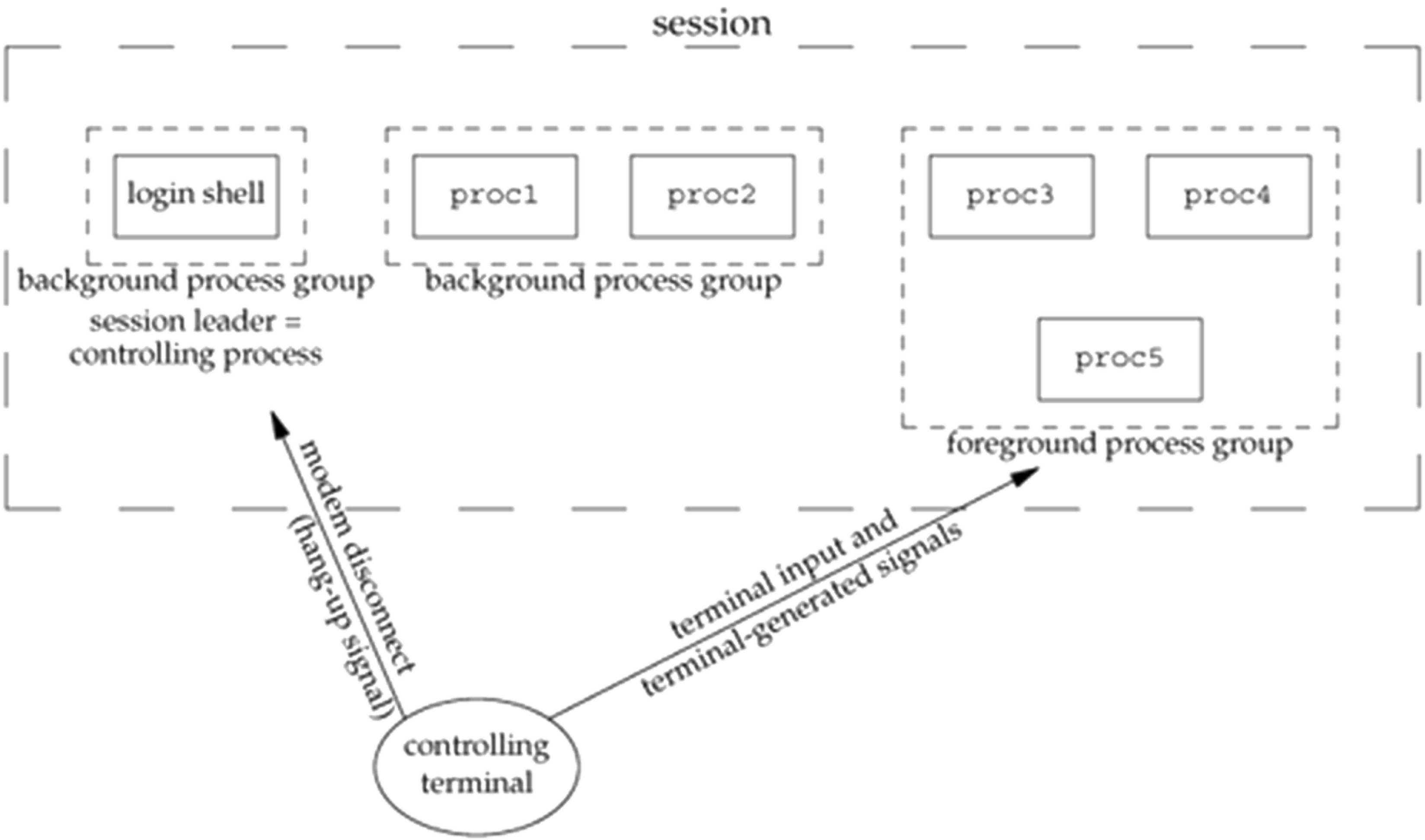
Controlling Terminal (2/3)

User control keys

- Send signals to all processes in the foreground process group
- Interrupt key (often Ctrl-C): Send SIGINT
- Quit key (often Ctrl-Backspace): Send SIGQUIT

If a network disconnect is detected by the terminal interface, the SIGHUP is sent to the controlling process (the session leader)

Controlling Terminal (3/3)



Whom to Send Signals?

How does the terminal device know the foreground process group?

It can be set using the tcgetpgrp and tcsetpgrp functions

Synopsis

- `pid_t tcgetpgrp(int filedes);`
- `int tcsetpgrp(int filedes, pid_t pgrpid);`

It can be only set by the controlling process, who knows the descriptor of the controlling terminal

Most applications don't call these two functions directly

They are normally called by job-control shells

Direct Access to the Controlling Terminal

Usually, a controlling terminal is established automatically when we log in

There are times a program wants to talk to the controlling terminal directly

- For example, ask a user to input his/her password from the terminal even if the standard input or standard output is redirected

This can be done by opening the file /dev/tty

- This special file is a synonym within the kernel for the controlling terminal
- If the program doesn't have a controlling terminal, the open of this device will fail

Direct Access to the Controlling Terminal, an Example

```
#include <unistd.h>
#include <stdio.h>
int main() {
    FILE *fp;
    if((fp = fopen("/dev/tty", "w")) == NULL) {
        fprintf(stdout, "cannot open the controlling terminal.\n");
        return(-1);
    }
    fprintf(fp, "write to /dev/tty\n");
    fprintf(stdout, "write to stdout\n");
    return(0);
}
```

```
$ ./a.out
write to /dev/tty
write to stdout
$ ./a.out > xxx
write to /dev/tty
$ cat xxx
write to stdout
```

Job Control

This feature allows us to start multiple jobs from a single terminal

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

Job Control (Cont'd)

Start a job in background – the & operator

```
$ ps auxw | grep ps &
[1] 30554
$ chuang 30553 0.0 0.0 2744 1016 pts/1 R 12:26 0:00 ps auxw
chuang 30554 0.0 0.0 3240  812 pts/1 S 12:26 0:00 grep ps
```

(Just press enter)

```
[1]+  Done                    ps auxw | grep ps
```

Stop a job running in foreground

- A user can press Ctrl-Z to stop a running foreground job
- The SIGTSTP is sent to all processes in the foreground process group

SIGTTIN and SIGTTOU

Processes in the foreground process group is always able to read from and write to the terminal

However, background processes is restricted to do so

An example of reading from the terminal – received SIGTTIN

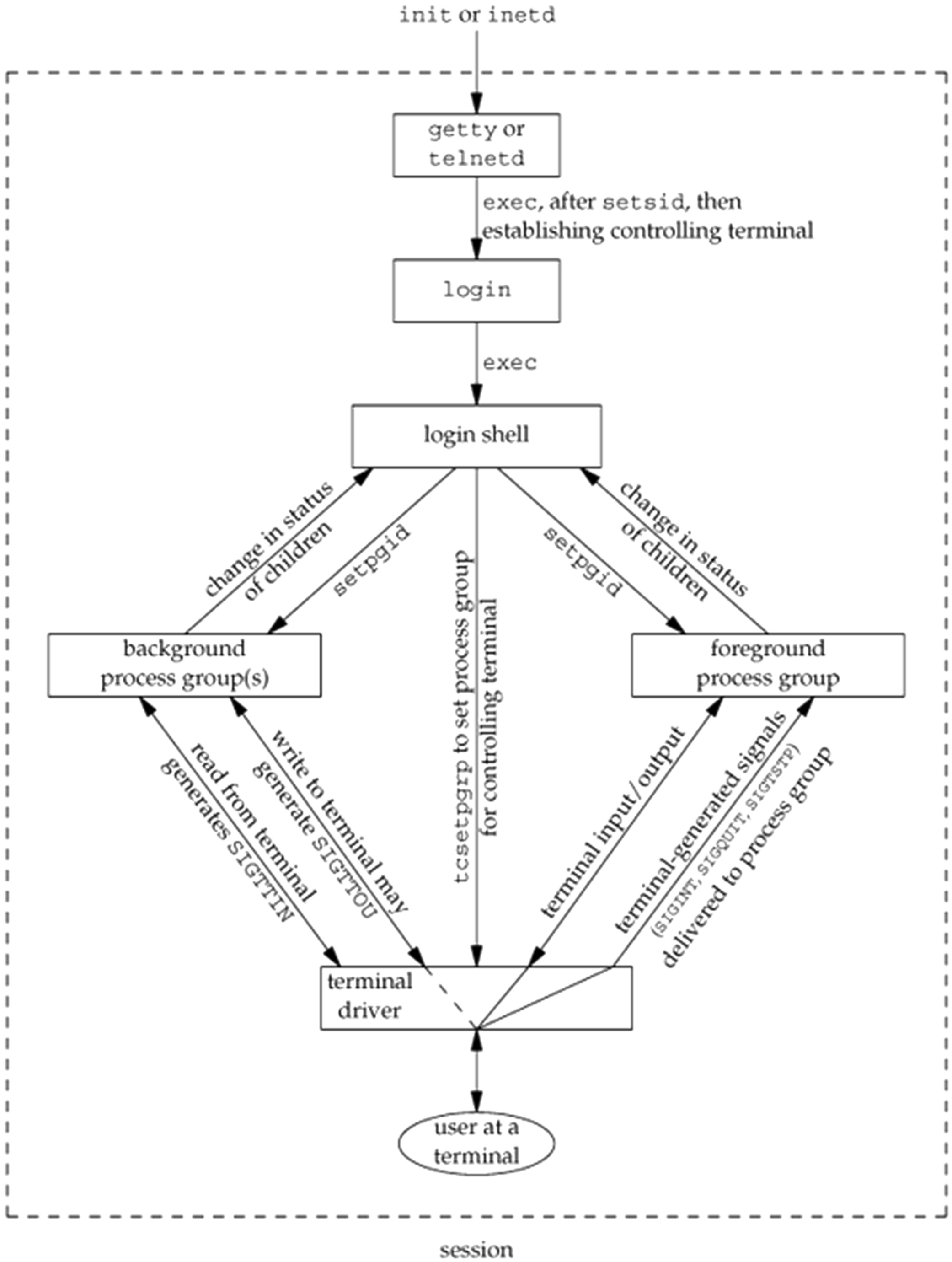
`$ cat > temp.foo &`	*start in background, but it'll read from standard input*
`[1] 1681`	
`$`	*we press RETURN*
`[1] + Stopped cat > temp.foo`	
`$ fg %1`	*bring job number 1 into the foreground*
`cat > temp.foo`	*the shell tells us which job is now in the foreground*
`hello, world`	*enter one line*
`^D`	*type the end-of-file character*
`$ cat temp.foo`	*check that the one line was put into the file*
`hello, world`	

SIGTTIN and SIGTTOU (Cont'd)

An example of reading from the terminal – received SIGTTOU

```
$ 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 the 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
```

Summary of Job Control Features



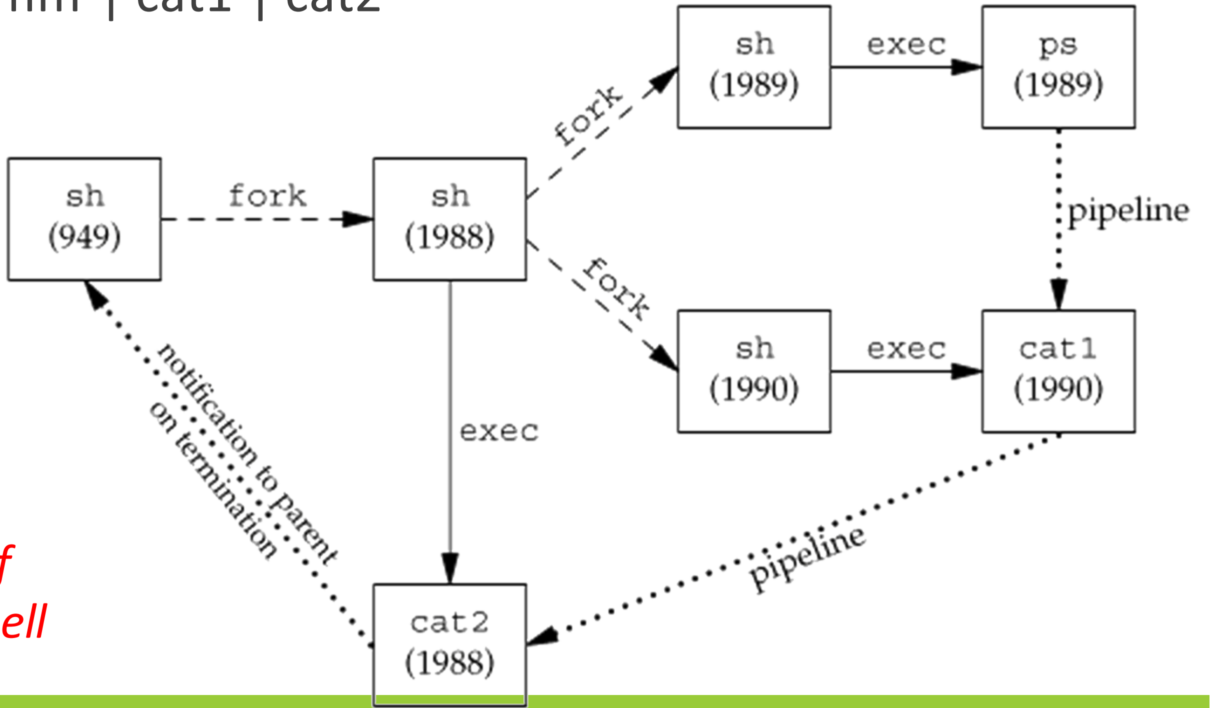
Shell Execution of Programs

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

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

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

```
 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
```



**** This example comes from the textbook and it might be different if you are working with a different shell*

Shell Execution of Programs (Cont'd)

bash 4.3.11 @ Ubuntu 14

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

```
  PID  PPID  PGID   SID COMMAND
19064 19060 19064 19064 bash
19237 19064 19237 19064 ps
```

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

```
  PID  PPID  PGID   SID COMMAND
19064 19060 19064 19064 bash
19238 19064 19238 19064 ps
19239 19064 19238 19064 cat1
19240 19064 19238 19064 cat2
```

Orphaned Process Groups

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

A entire process group can be orphaned

Definition of an orphaned process group

- A process group is orphaned if the parent process of every member is either a member of the group or not a member of the group's session
- In contrast, a process group is not orphaned if a process in the group has a parent in a different process group but in the same session

If a process group becomes orphaned

- Every stopped process in the group is sent the SIGHUP followed by the SIGCONT
- The default action on receipt of a SIGHUP is to terminate the process

Orphaned Process Group, an Example

```
main(void) {
    char    c;
    pid_t   pid;
    pr_ids("parent");                         /* parent: pid, ppid, pgrp, and tpgrp */
    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");                      /* child: pid, ppid, pgrp, and tpgrp */
        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);
    }
}
```

Orphaned Process Group, an Example (Cont'd)

```
$ ./fig9.11-orphan3
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
```
(sleep for 5 seconds)
```
SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error from controlling TTY, errno = 5
```

The parent and the child prints out their own information

The parent then sleeps for 5 seconds

The child stopped itself

When the parent terminates, the child received SIGHUP and SIGCONT

- Since the child has assigned the SIGHUP handler, it is not terminated
- The child is now in background, so read from TTY got the EIO error

Q & A