



Process Relationships

- Terminal and Network Logins
- Process Groups and Sessions
- Job Control
- Relationships

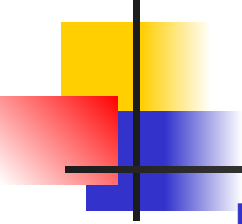


Terminal Logins

- Terminals
 - Point to Point connection
 - Local – hard wired
 - Remote – direct connection via modem

- Dumb terminals connected to host(Host had a fixed number of these terminal devices, known upper limit)

Evolution of terminals

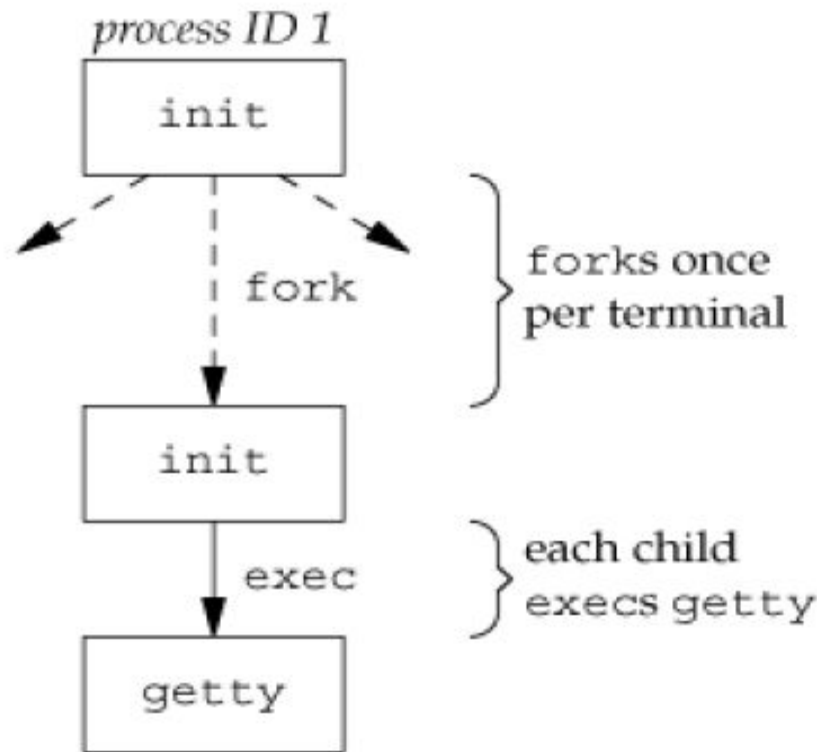
- 
-
- Point to point
 - Terminal windows
 - Windowing system(user can start after logging or automatically started)
 - Character based terminal
 - Emulated Graphical terminal
 - Graphical terminal running a windowing system.

Terminal Login Process



- init reads terminal configuration file `/etc/ttys` (`/etc/inittab` for Linux) – one line per terminal device – specifies name of the device and other parameters required for `getty` program.
- `init` process reads the `/etc/ttys` file and forks and execs `getty` for each terminal
- `getty` opens terminal device for reading and writing and sets up standard file descriptors 0, 1 and 2
- `getty` prompts for username (`login:` and waits for us to enter our user name) and execs `login`
`execle("/bin/login", "login", "-p", username, (char *) 0, envp);`

Processes invoked by init to allow terminal logins



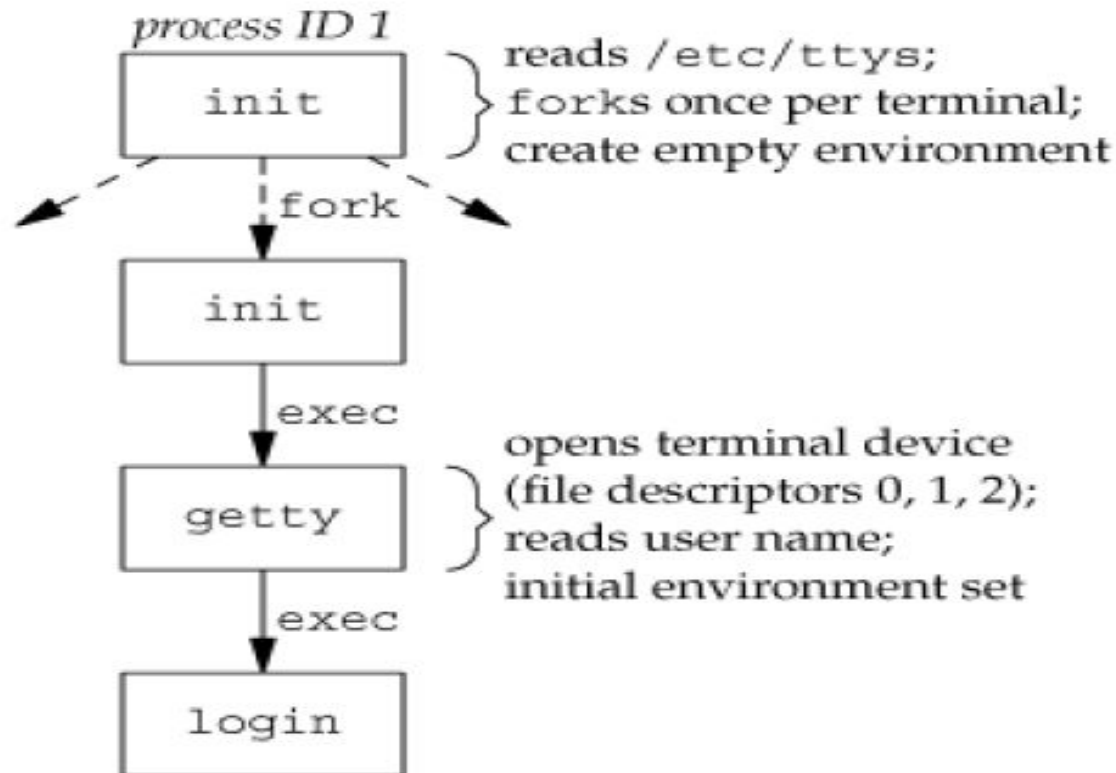
All the processes have a real user ID of 0 and an effective user ID of 0 (they all have superuser privileges). The init process also execs the getty program with an empty environment

Terminal Login Process



- When we enter our user name, getty's job is complete, and it then invokes the login program i.e. getty execs login
`execle("/bin/login", "login", "-p", username, (char *) 0, envp);`

State of processes after `login` has been invoked



Note: The process ID of the bottom three processes is the same, since the process ID does not change across an `exec`. Also, all of them other than the original `init` process have a parent process ID of 1



Terminal Login Process

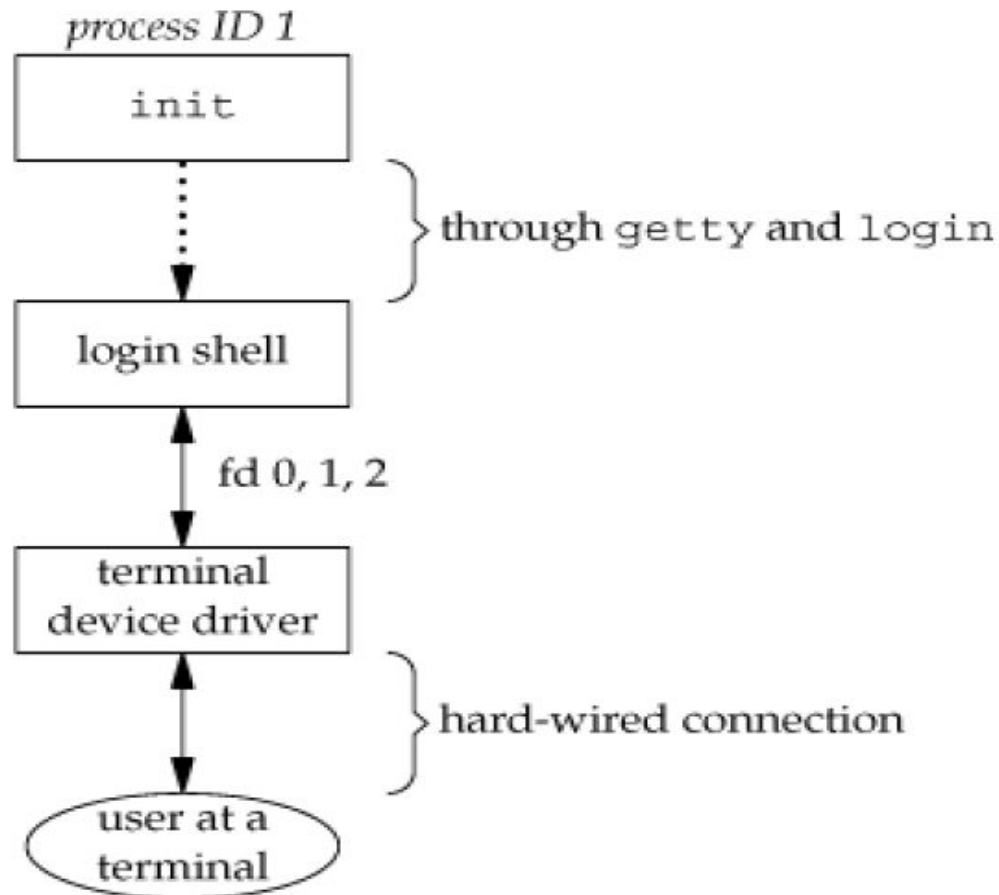
- login retrieves encrypted password stored in shadow file using `getpwnam` function.
- login calls `getpass` function to prompt for password. It then encrypts it by calling `crypt` function and compares result to what it retrieved from the shadow file
- If not successful, login will exit with status 1 and `init` will fork a new `getty` process for that terminal



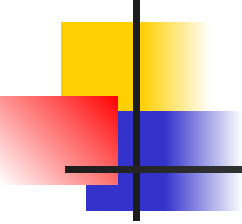
Terminal Login Process

- If successful login will
 - Change to user's home directory (chdir)
 - Change the terminal device ownership to the user (chown) so that we can read/write it
 - Set group IDs (setgid and initgroups)
 - Setup some of the environment variables
 - Set user IDs (setuid)
 - Exec our default shell
`execl("/bin/sh", "-sh", (char *)0);`

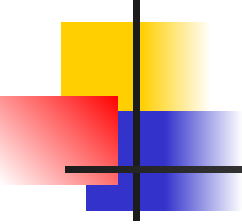
Arrangement of processes after everything is set for a terminal login



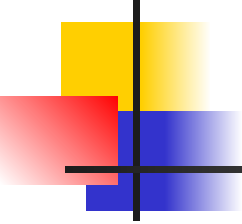
Network Logins

- 
- Unknown, possibly large number of connections. Not practical to fork a getty for each possible one
 - 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.

Network Logins

- 
- 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 (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.

Network Logins

- 
- 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
 - To summarize,
 - Wait for an incoming network connection, and create a process as needed.
 - Pseudo-terminal is a converter acts like a terminal to the rest of the system but can communicate with the network as well.



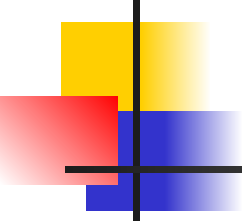
Network Login Process

- Internet Supervisor – inetd process. When system starts up, init forks a shell that runs commands in /etc/rc shell script.
- One of those commands is a command to start the inetd process. Once the shell exits, the inetd process is inherited by init
- inetd listens for TCP/IP connection requests and forks/execs the appropriate program

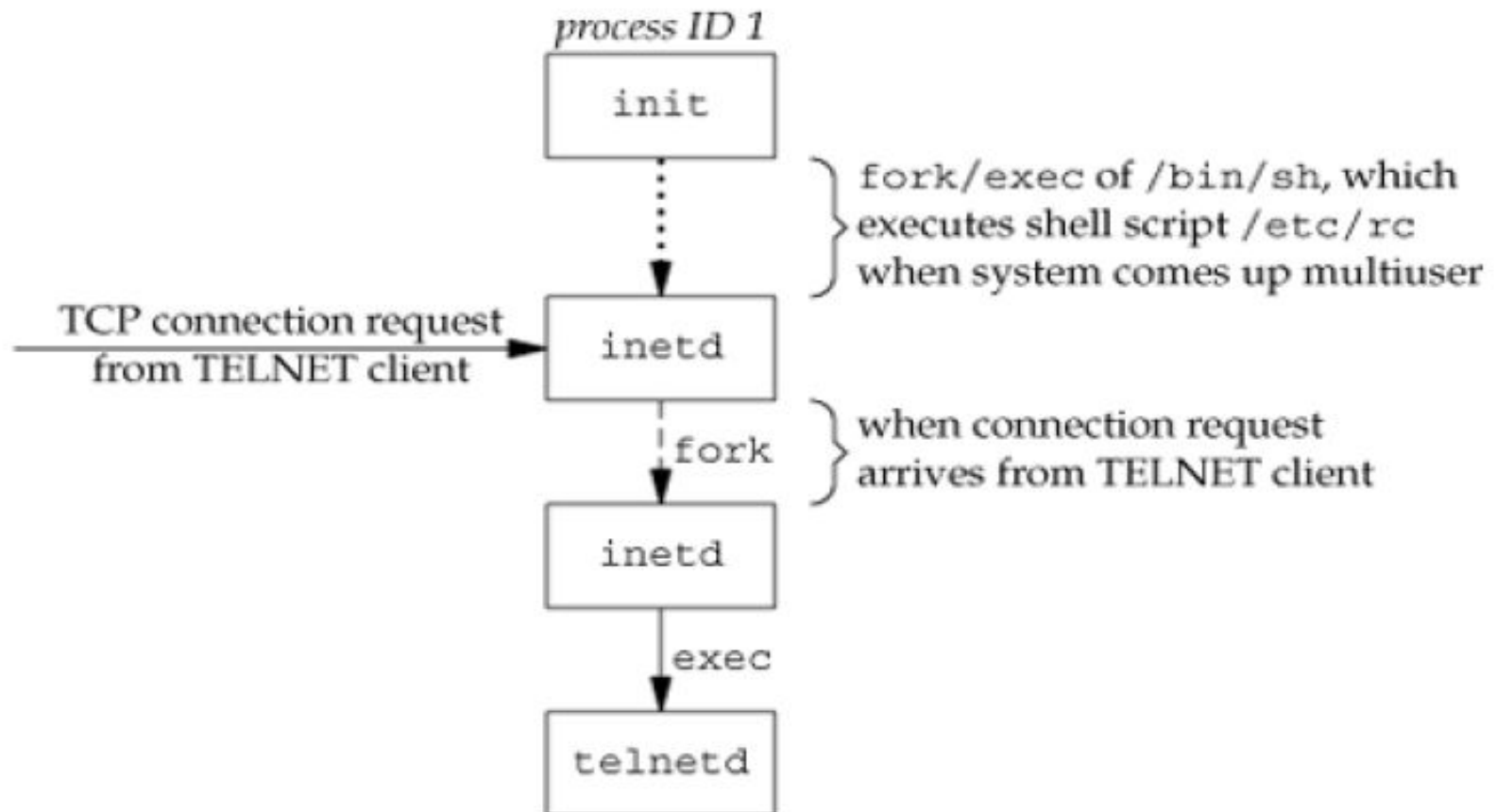


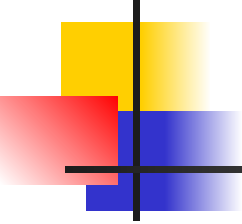
Network Login Process

- Example, Let a TCP connection request arrives for the TELNET server.
- Note: 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`

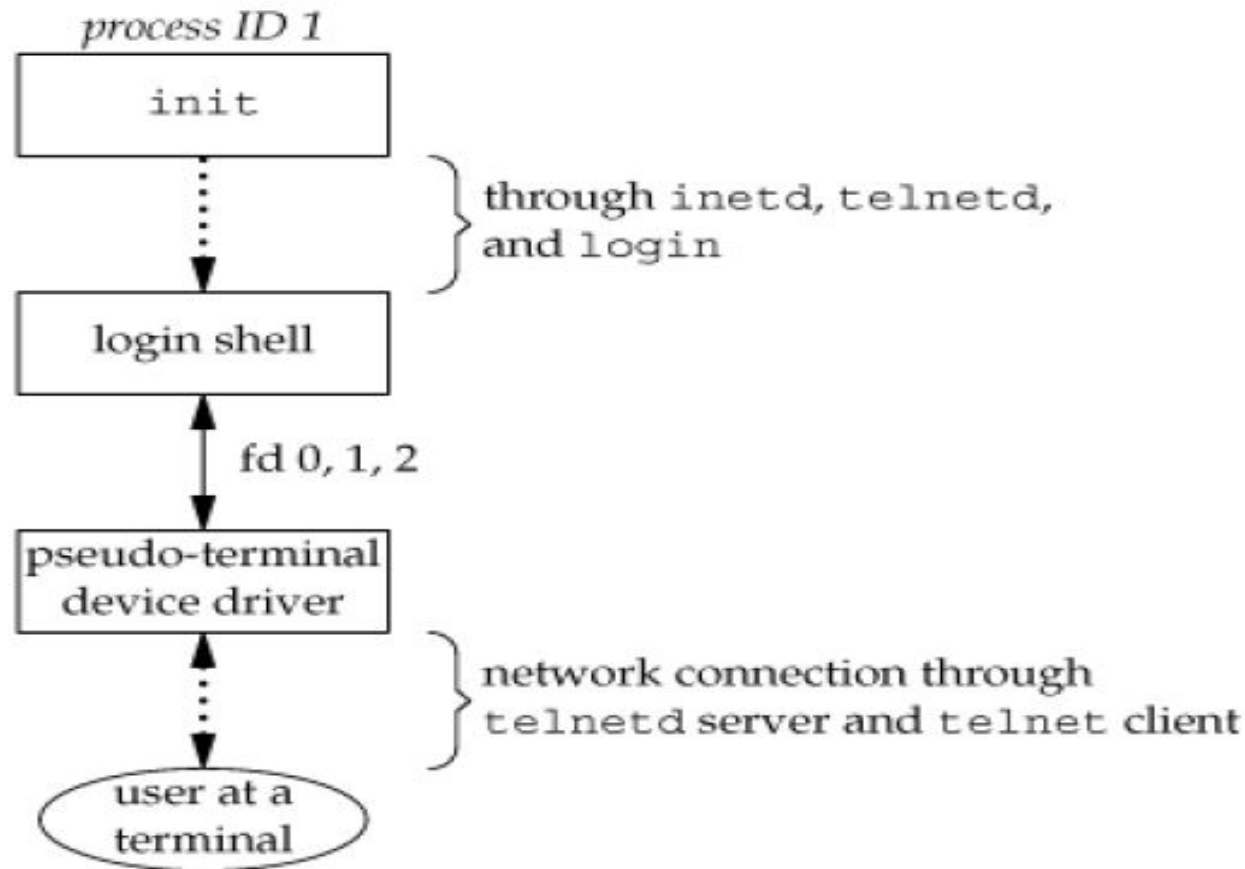
- 
-
- The client opens a TCP connection to *hostname*, and the program that's started on *hostname* is called the TELNET server.
 - The client and the server then exchange data across the TCP connection using the TELNET application protocol.
 - The user who started the client program is now logged in to the server's host

Sequence of processes involved in executing TELNET server



- 
-
- In the case of TELNET, inetd forks and execs telnetd
 - telnetd opens a pseudo terminal device and forks a child
 - Parent handles network communication
 - Child execs login
 - Parent and 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 as described in terminal login

Arrangement of processes after everything is set for a network login





Network logins

- Note 1: A lot of things happen between the pseudo-terminal device driver and the actual user at the terminal.
- Note 2: Whether we log in through a terminal or a network, 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
- Note 3: Linux uses extended Internet services daemon, xinetd.



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, 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_t` data type.

Process Groups



```
pid_t getpgrp(void);
```

Returns: process group ID of calling process

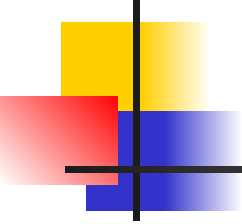
```
pid_t getpgid(pid_t pid);
```

Returns: process group ID if OK, 1 on error

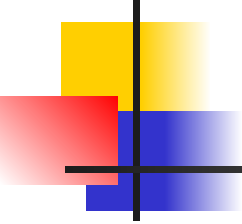
- Gets the process group ID of the current process or the specified process
- Note: If *pid* is 0, the process group ID of the calling process is returned. Thus,

`getpgid(0);` is equivalent to `getpgrp();`

Process Groups

- 
-
- 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.

Process Groups

- 
- The process group 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.



Process Groups

- 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
- 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).
- The `waitpid` function lets us wait for either a single process or one process from a specified process group.



Process Groups

- 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.
- Note: A process can set the process group ID of only itself or any of its children. It can't change the process group ID of its child that has called one of the exec functions.



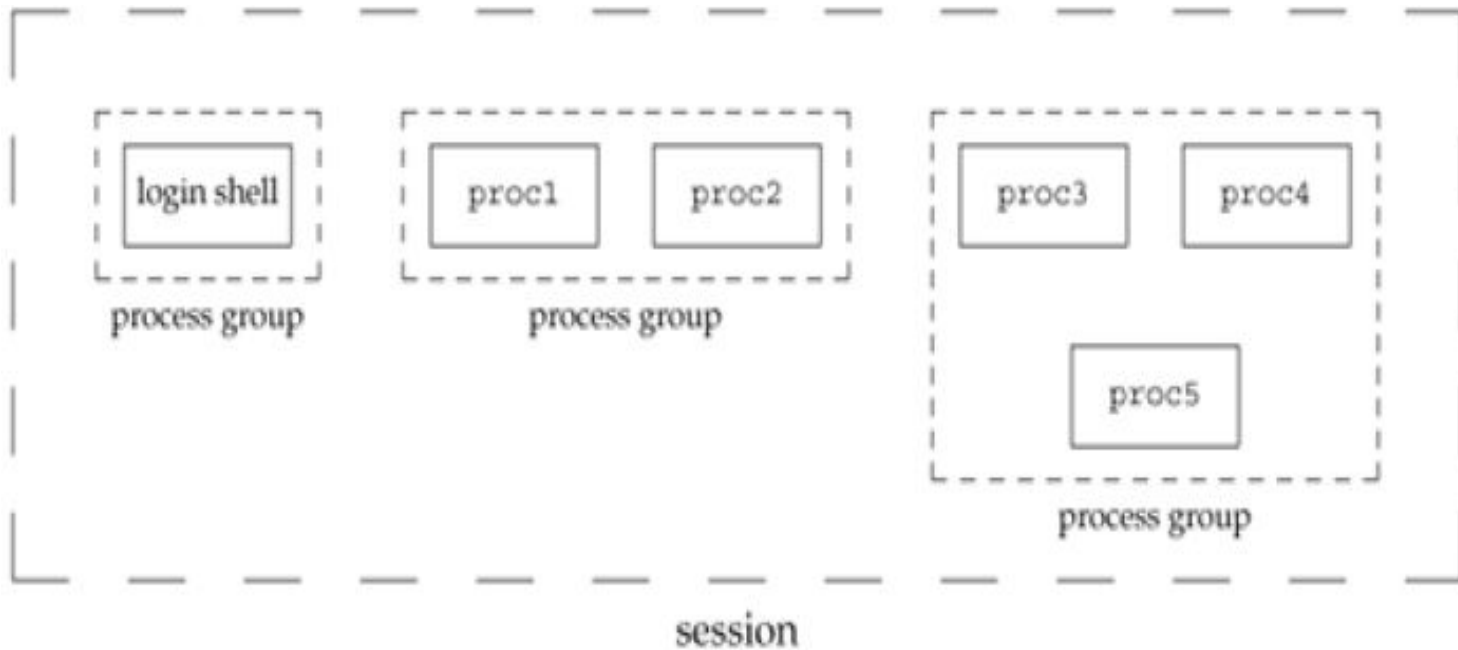
Process Groups

- In most job-control shells, the `setpgid` 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.



Sessions

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





Creating A New Session

`pid_t setsid(void)`

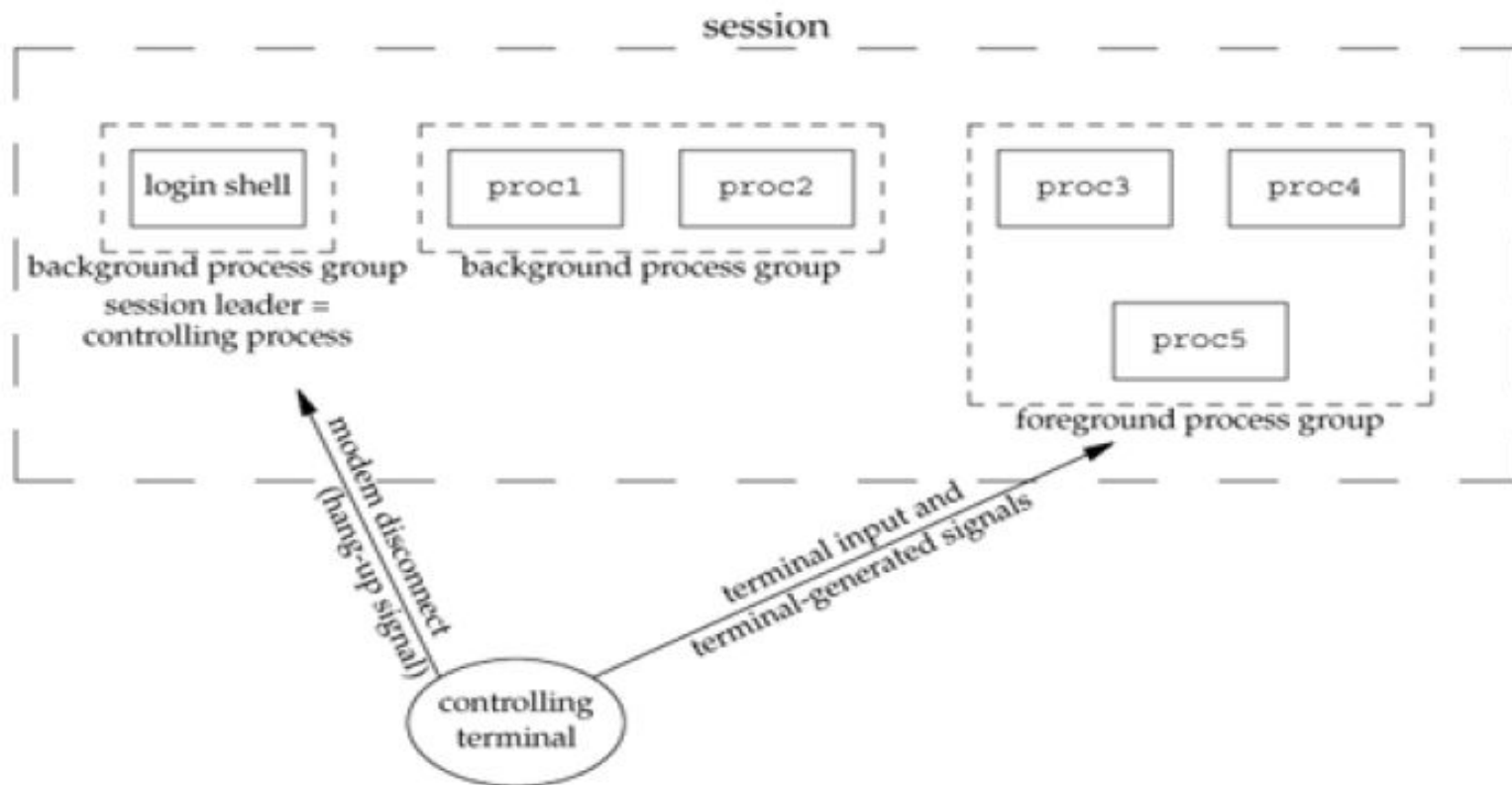
- If process is already a process group leader, returns -1
- If process is not already a process group leader
 - A new session is created, and the calling process is its leader
 - A new process group is created with the calling process as its leader
 - Process has no controlling terminal. If it already had one, the connection is broken



Controlling Terminal

- Sessions can have at most one controlling terminal (usually the terminal or pseudo terminal that we log in from)
- Session leader that opens the terminal is called the controlling process
- If the session has a controlling terminal then it has one foreground process group and may have several background process groups
- Foreground process group receives terminal input
 - SIGINT Ctrl-C
 - SIGQUIT Ctrl-\
 - SIGTSTP Ctrl-Z
- Network (or modem) disconnect causes a signal (SIGHUP) to be sent to the controlling process (session leader)

Controlling Terminal

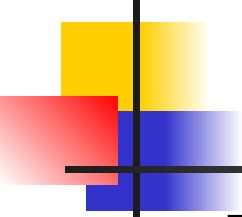




Job Control

- Allows us to run multiple process groups (jobs) simultaneously
- Must be able to control which process is the foreground process and be able to stop and continue processes
- Job control requires
 - A shell that supports it (Bourne doesn't)
 - Terminal driver supports it
 - Kernel supports certain job-control signals

Job Control

- 
- By 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.

Job Control

- When we start a background job, the shell assigns it a job identifier and prints one or more of the process IDs.

```
$ make all > Make.out &
```

```
[1]      1475
```

```
$ pr *.c | lpr &
```

```
[2]      1490
```

```
$
```

just press RETURN

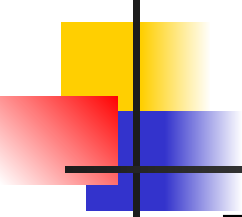
```
[2] + Done
```

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

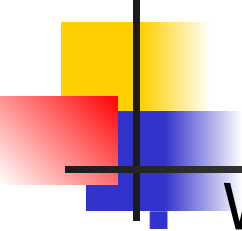
```
[1] + Done
```

```
make all > Make.out &
```

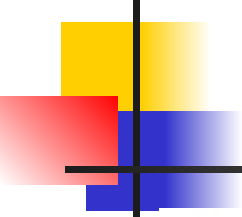
Job Control

- 
-
- 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 when we press RETURN, the shell tells us that the jobs are complete.
 - We have to press RETURN 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.

Job Control

- 
-
- 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 from using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal.

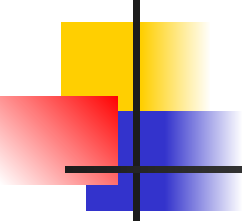
Job Control



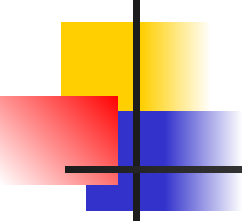
```
$ cat > temp.foo &           start in background, but it'll read from standard input
[1]      1681
$                               we press RETURN
[1] + Stopped (SIGTTIN)       cat > temp.foo &
$ fg %1                       bring job number 1 into the foreground
cat > temp.foo                the shell tells us which job is now in the foreground
```

- 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.

Job Control

- 
- The shell detects this change in status of its and tells us that the job has been stopped.
 - We then move the stopped job into the foreground with the shell's `fg` command.
 - Doing this causes the shell to place the job into the foreground process group 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.

Job Control

- 
-
- 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.

Job Control



```
$ 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

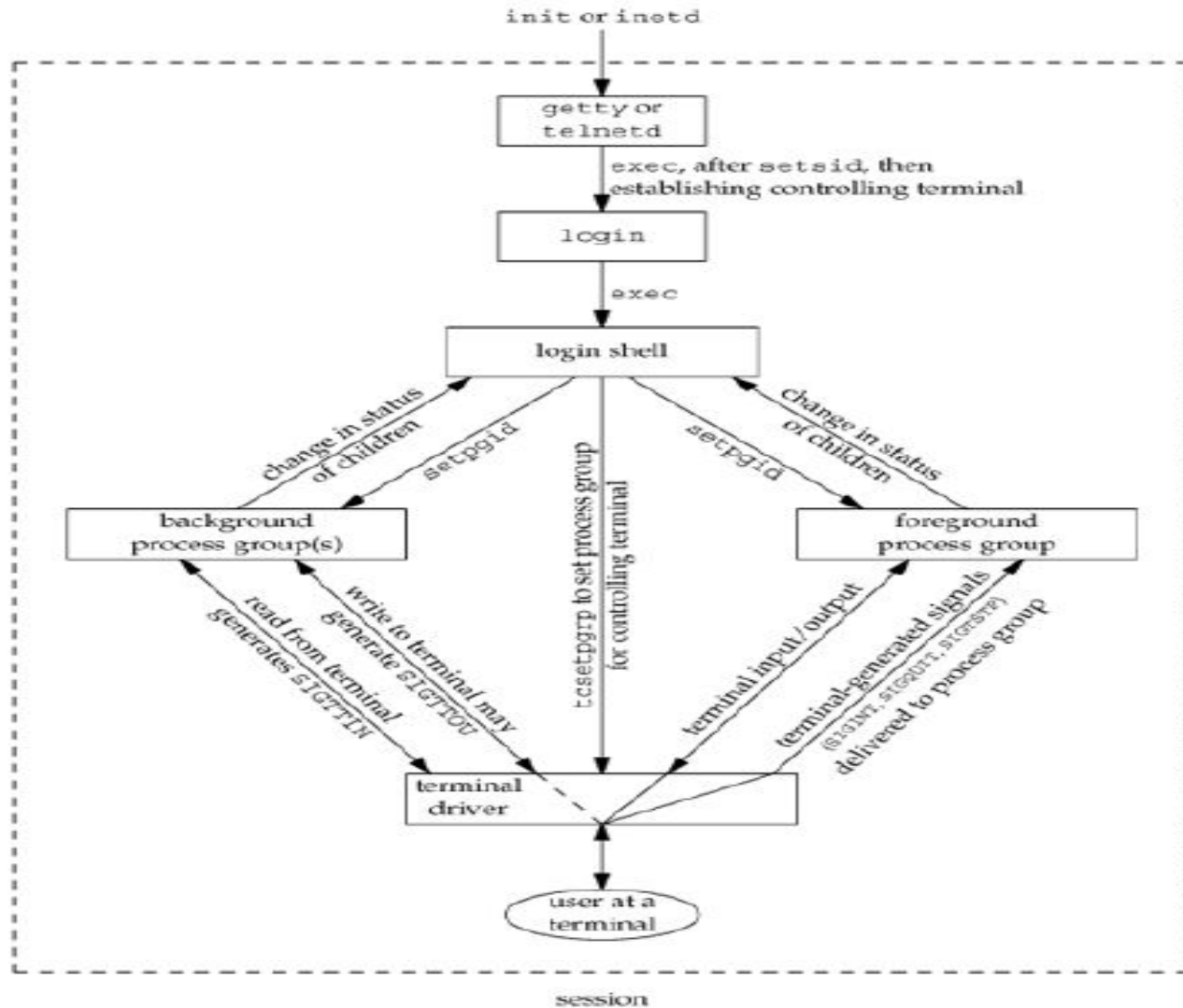
```
cat temp.foo
```

the shell tells us which job is now in the foreground

```
hello, world
```

and here is its output

Job Control





Shell Execution of Programs

- When we type in a command the shell does a fork and exec
 - If it's a foreground process the parent waits
 - If it's a background process the parent continues
- Pipeline – ex: `ls | grep "hello"`
 - Process order depends on shell
 - Often last command is forked first



Examples

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

PID	PPID	PGID	SID	TPGID	COMMAND
28398	28397	28398	28398	28478	bash
28478	28398	28478	28398	28478	ps



Examples

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

PID	PPID	PGID	SID	TPGID	COMMAND
28398	28397	28398	28398	28398	bash
28509	28398	28509	28398	28398	ps



Examples

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

PID	PPID	PGID	SID	TPGID	COMMAND
28398	28397	28398	28398	28517	bash
28517	28398	28517	28398	28517	ps
28518	28398	28517	28398	28517	cat



Examples

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

PID	PPID	PGID	SID	TPGID	COMMAND
28398	28397	28398	28398	28398	bash
28542	28398	28542	28398	28398	ps
28543	28398	28542	28398	28398	cat