Lecture 12. Process Relationships

So far, we've seen that every process has a parent process. The parent receives a termination status when the child terminates. The `waitpid()` function enables the parent to wait for a child to terminate and obtain its status.

This lecture deals with **process groups**—groups of processes that execute together. It references the concept of signals, which is the subject of the next lecture. For now, we just mention that signals are interrupts issued by processes to each other, in order to cause the execution of certain actions in response to an erroneous situation that the signal is "signaling" and the receiver of the signal is "catching".

The most commonly encountered "erroneous situations" (program faults) are illegal memory references, execution of peculiar instructions, or floating point errors. The most common user-issued signals are interrupt, quit, hangup, and terminate. When one of these occurs, the signal is sent to all processes that were started from the same terminal, and all are terminated. A core image file is written for debugging.

Terminal Logins

The **login procedure** starts with a file containing names, one per line, of terminal devices and related parameters that are to be passed to the `getty()` program. This file historically resided (and still does) in `/etc/ttys`.

From the `man` pages on my system, here are the fields in this file:
- The first field is normally the name of the **terminal special file** as it is found in `/dev`. However, it can be any arbitrary string when the associated command is not related to a tty.

- The second field of the file is the **command to execute for the line**, usually `getty(8)`, which initializes and opens the line, setting the speed, waiting for a user name and executing the `login(1)` program. It can be, however, any desired command, for example the start up for a window system terminal emulator or some other daemon process, and can contain multiple words if quoted.

- The third field is the **type of terminal usually connected to that tty line**, normally the one found in the `termcap(5)` data base file. The environment variable TERM is initialized with that value by either `getty(8)` or `login(1)`.

- The remaining fields set flags in the `ty_status` entry (see `getttyent(3)`), specify a window system process that `launchd(8)` will maintain for the terminal line.

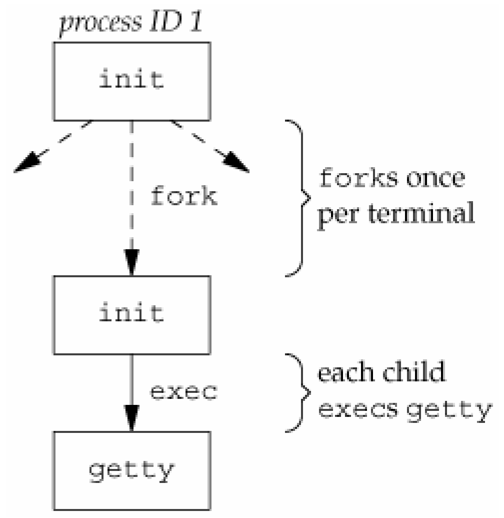
Entries from the `/etc/ttys` file:

```
tty.serial      "/usr/libexec/getty serial.57600"       vt100
off secure

fax     "/usr/bin/fax answer"   unknown off

tty[00-07]      "/usr/libexec/getty std.9600"   unknown off
secure
```

The structure `ttyent` from `ttyent.h` specifies the fields in `/etc/ttys`:

```
struct ttyent
{
     char    *ty_name;       /* terminal device name */
     char    *ty_getty;      /* command to execute, usually
getty */
     char    *ty_type;       /* terminal type for termcap */
#define TTY_ON          0x01    /* enable logins (start ty_getty
program) */
#define TTY_SECURE      0x02    /* allow uid of 0 to login */
#define TTY_DIALUP      0x04    /* is a dialup tty */
#define TTY_NETWORK     0x08    /* is a network tty */
     int     ty_status;      /* status flags */
     char    *ty_window;     /* command to start up window
manager */
     char    *ty_comment;    /* comment field */
     char    *ty_group;      /* tty group name */
};
```

When the system is bootstrapped, the kernel creates process ID 1, namely `init`, which reads the file `/etc/ttys` and does a `fork()` and an `exec()` of the program `getty` for every terminal in this file that allows a login. The following diagram shows the resulting processes: each terminal gets a copy of `init` (with `fork()`), and each `init` execs `getty` in the child (with `exec()`).



Then `getty` calls `open()` for the terminal. Thus the terminal is open for reading and writing. File descriptors 0 (stdin), 1 (stdout), and 2 (stderr) are set to the terminal. Then getty outputs a login prompt (after detecting the baud rate [bits/s] of the terminal and adapting it).

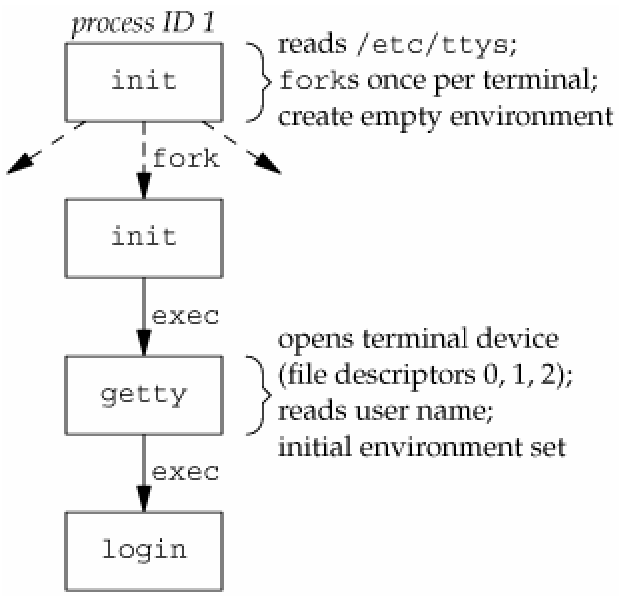
So `getty [type [tty]]` gets called by `launchd(8)` to open and initialize the tty line, read a login name, and invoke `login(1)`.
Arguments:

- `tty`—is the special device file in `/dev` to open for the terminal (for example, `ttyh0`). If there is no argument or the argument is `-`, the tty line is assumed to be open as file descriptor 0.
- `type`—can be used to make `getty` treat the terminal line specially. This argument is used as an index into the `gettytab(5)` database, to determine the characteristics of the line. If there is no `/etc/gettytab` a set of system defaults is used. If indicated by the table, `getty` will clear the terminal screen, print a banner heading, and prompt for a login name. Usually either the banner or the login prompt will include the system hostname.

Initially, `init` invokes getty with an empty environment. However, getty retrieves from `/etc/gettytab` environment strings such as TERM.

All the processes started (`init`, `getty`, `login`) have superuser privileges. The process IDs of these 3 processes (one group per terminal) is the same, since the process ID does not change across calls to `exec()`. Also, since they were all forked by init, their parent process ID is 1.

The process diagram up to and including the call to `login` is as follows:



`login` does the following:

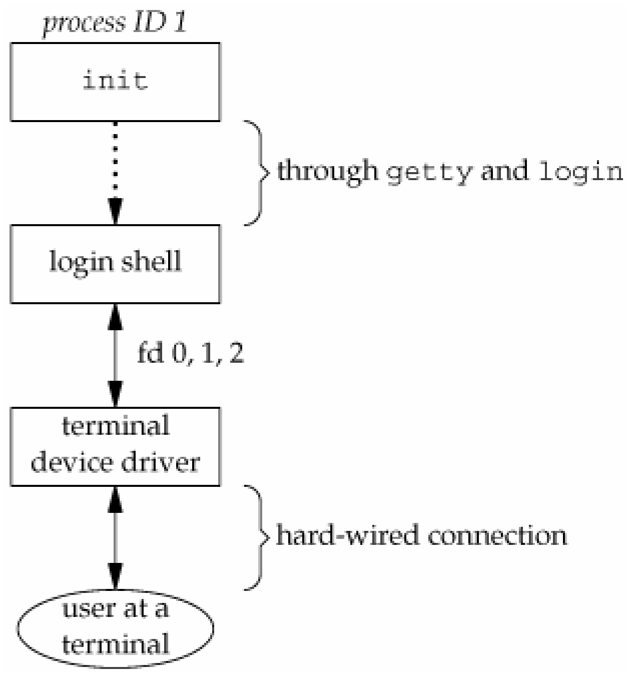
- captures username when entered at the prompt;
- calls the `getpwnam()` system call to retrieve the stored password associated with that username in the password file;
- calls `getpass(3)` to display the password prompt and read the entered password;
- calls `crypt(3)` to encrypt the entered password and to compare that with the value returned by `getpwnam()` (the `pw_passwd` entry from the password file);
- calls the `exit()` system call with an argument of 1 if a number of failed logins are attempted. This termination is signaled to `init`, which starts the process all over again for this terminal;
- calls the `chdir()` system call to change to user's home directory if the login is correct;
- calls the `chown()` system call to set user as owner and user's group as group owner of this terminal;
- sets access permissions for the terminal to have user read, user write, and group read enabled;
- calls `setgid()` to set the user group ID and `initgroups()` to initialize the group access list;
- initializes the user's environment: HOME, SHELL, USER, LOGNAME, PATH;

- calls `setuid()` to change user ID to user's ID;
- calls `exec()` to invoke user's login shell, for example:

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

- optionally, `login` also prints the message-of-the-day, checks for new mail etc.

After the successful call to the shell program, the process diagram has grown to:



The login shell reads the start-up files (`.bash_profile` for bash, for example). The start-up files change or add to the environment variables set by `login`.

SVR4 supports two forms of terminal logins:
- `getty`—for the console
- `ttymon`—for other terminal logins

`ttymon` is part of SAF (Service Access Facility). The sequence is slightly different: `init` forks and its copy execs `ttymon` (instead of `getty`). `ttymon` monitors all the terminal ports listed in its configuration file and forks when the user enters his/her username. `ttymon` forks and its copy execs `login`, from which point the sequence of calls is the same as for `getty`: `login` ultimately calls the shell program. The only difference is that now, the shell's parent is `ttymon` (instead of `init`).

Network Logins

In the case of multiple hardwired terminals, for the terminal login process, `init` knows in advance by parsing the `ttys` file how many terminals there are, and spawns (forks and its copy execs) a `getty` process for each one.

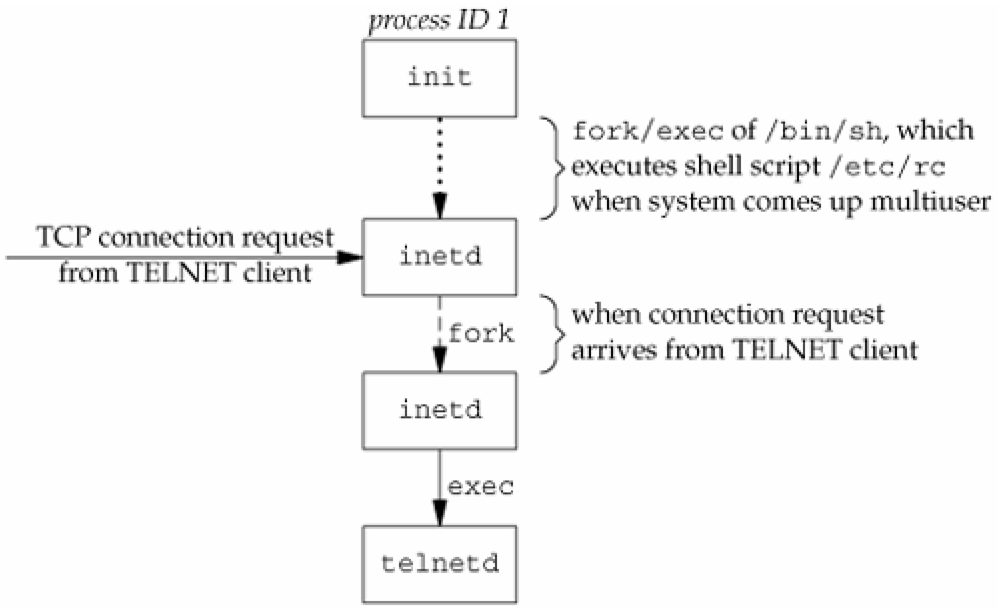
In contrast, in the case of network logins, the process that waits for network connection requests (as opposed to hardwired terminal requests) is `inetd` (not `getty`).

The sequence of events is as follows:

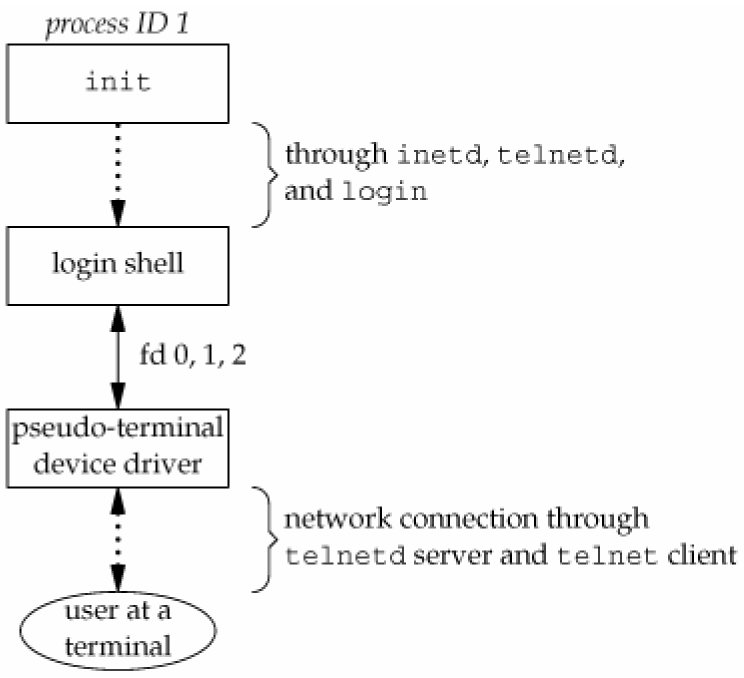
- `init` (as part of the system start-up) invokes a shell that executes the shell script `/etc/rc`. This starts several daemons (programs that run permanently), among which `inetd`. The parent process of `inetd` is `init`;
- `inetd` waits for TCP/IP connection requests, then forks and its copy execs the appropriate program;
- assume a TCP connection request has arrived for the TELNET server, running on "this" host. The request has been sent by a user on the remote host (or even on the same host). The user initiates the login by starting the TELNET client on the remote host:

```
$ telnet hostname
```

The client and server exchange data across the TCP connection using the TELNET protocol. The user on the remote host is now logged into the server's host. The sequence of processes is shown in the following diagram:


process ID 1
init
fork/exec of /bin/sh, which executes shell script /etc/rc when system comes up multiuser
TCP connection request from TELNET client
inetd
fork
when connection request arrives from TELNET client
inetd
exec
telnetd

telnetd opens a pseudo-terminal device and forks. The parent handles the communication across the network connection, while the child execs the login program. The child sets up file descriptors 0, 1, and 2 to the pseudo-terminal. login sets up the user's home directory, user's group ID and user ID, and the user's initial environment. login then execs the shell in exactly the same way as for the local terminal logins.


process ID 1
init
through inetd, telnetd, and login
login shell
fd 0, 1, 2
pseudo-terminal device driver
network connection through telnetd server and telnet client
user at a terminal

Process Groups

Each process has a process ID and a process group ID. A **process group** is a collection of one or more processes. Process group IDs are positive integers that are stored in a pid_t type. The function getpgrp() returns the process group ID:

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

pid_t getpgrp(void);
```

Returns process group ID of calling process.

Each process group has a **process group leader**. The leader lends its process ID as the process group ID. The process group still exists, in spite of the leader having terminated, as long as there is at least one process in the group. The period of time between the moment when the group was created and the moment when the last remaining process terminates is called the **process group lifetime**. Processes from the group may either terminate, or join other process groups.

A process enters a group or creates one by calling `setpgid()`:

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

int setpgid(pid_t pid, pid_t pgid);
```

Returns 0 if OK, -1 if error.

This sets the process group ID to the argument passed in, *`pgid`*. If, in addition, the two arguments are equal, i.e. the process ID and process group ID are equal, the process becomes leader of the group.

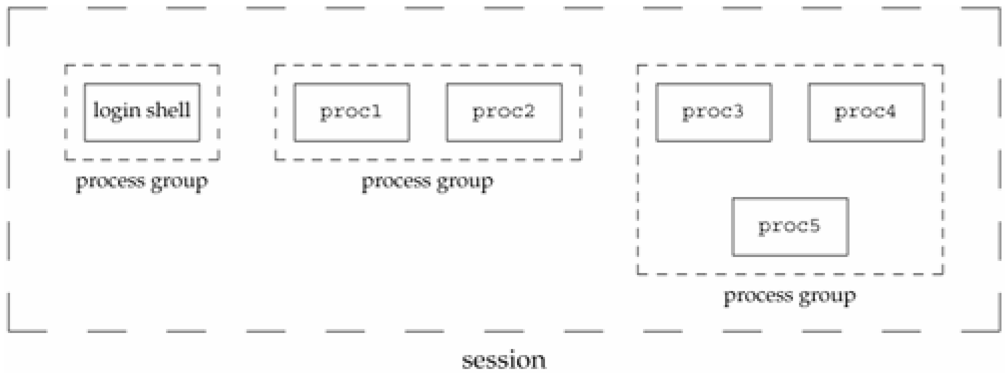
A process can set the process group ID of only itself or one of its children. Furthermore, it can only set the pgid of one of its children as long as they haven't called `exec()`.

A default value of *`pid`*`=0` sets the group ID to the process ID of the caller. A default value of *`pgid`*`=0` sets the group ID to *`pid`*.

Usually, `setpgid()` is called after a `fork()` to make sure the child has a set process group ID.

Sessions

A **session** is a collection of one or more process groups. For example, the following diagram shows a session formed with three process groups:



This arrangement could have been generated by the following shell command:

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

A process establishes a new session by calling the `setsid()` function:

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

pid_t setsid(void);
```

Returns process group ID if OK, -1 if error.

The calling process cannot be a process group leader; the function returns with error, in that case. If a process group leader needs to establish a process, it has to fork first (and have the child call `setsid()`, since the child will get a new process ID, even though it inherits the same process group ID).

When calling `setsid()`, three things happen:

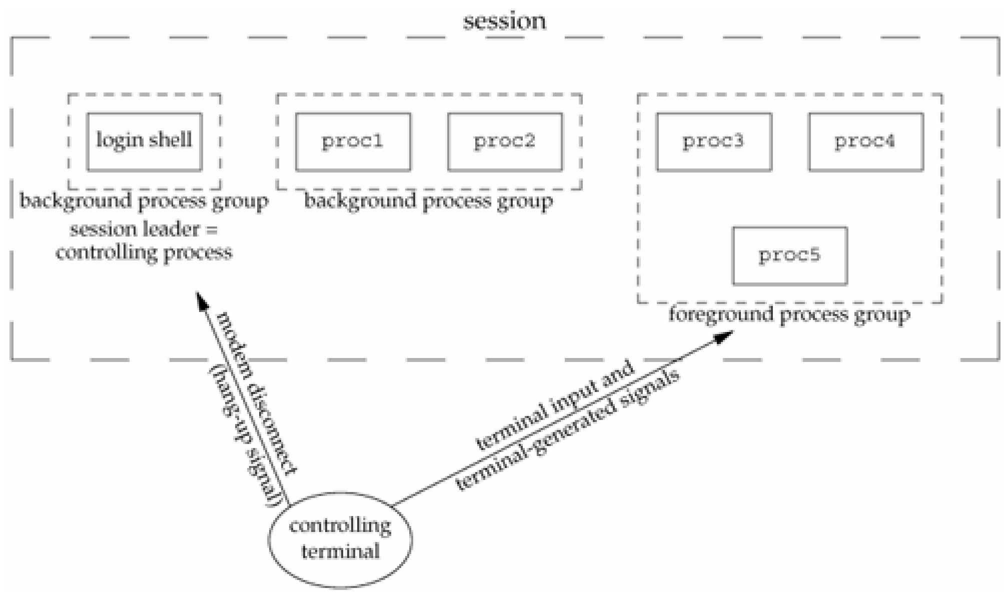
1. Caller becomes session leader;
2. Caller becomes process group leader. The new group gets a process group ID equal to caller's;
3. Caller has no controlling terminal. If it had one before calling `setsid()`, the association is broken.

Controlling Terminal

A **controlling terminal** is the terminal device (local terminal login) or pseudo-terminal device (network terminal login) associated to a login session.

Sessions and process groups have the following characteristics:

- a session can have a single controlling terminal;
- the session leader is called the controlling process;
- within a session, there is a single foreground process group, and there may be multiple background process groups;
- when the terminal's interrupt/quit key combination is pressed (Ctrl-C/Ctrl-\), an interrupt/quit signal is sent to all foreground processes;
- if a modem disconnect is detected by the terminal, a hang-up signal is sent to the controlling process (the session leader).



The `tcgetpgrp()` and `tcsetpgrp()` Functions

To tell the kernel which process group is foreground, and thus to let the terminal device driver know where to send the terminal input and terminal-generated signals, the function `tcgetpgrp()` is used to return the foreground progress group:

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

pid_t tcgetpgrp(int fd);
```
Returns the process group ID of the foreground process if OK, -1 if error.
```
int tcsetpgrp(int fd, pid_t pgrpid);
```
Returns 0 if OK, -1 if error.

The function `tcgetpgrp()` returns the process group ID of the foreground process group associated with the terminal open on *`fd`*.

The function `tcsetpgrp()` sets the process group ID of the caller to *`pgrpid`*, referring to its association to the controlling terminal identified by *`fd`*.

Job Control

Job control allows a user to start multiple jobs (groups of processes) from a single terminal and control which jobs can access the terminal and which jobs run in the background.

When starting a background job, the shell assigns it a job identifier and prints one or more of the process IDs. Example:

```
$ make myfilesharing.c &
[1] 973
$ make: Nothing to be done for `myfilesharing.c'.

[1]+  Done                    make myfilesharing.c
```

The shell will print the status of its background jobs only right before printing its prompt, to prevent further input from mixing with any unfinished output from the still running job.

The key combination Ctrl-Z suspends a foreground process and puts it in the background, by sending the SIGTSTP signal to all processes in the foreground process group.

To summarize, the following key combinations affect process groups (jobs):
- Ctrl-C (interrupt)—generates SIGINT
- Ctrl-\ (quit)—generates SIGQUIT
- Ctrl-Z (suspend)—generates SIGTSTP

The terminal driver must also handle the choice of job, among one foreground and multiple background, which would receive input from the terminal. The terminal sends the signal SIGTTIN to stop the background job. The shell brings it into the foreground so that the job can read from the terminal.

Shell Execution of Programs

The `ps` command displays process status information about all the running processes. It displays a header line, followed by lines containing information about all of the processes that have controlling terminals.

A different set of processes can be selected for display by using any combination of the `-a`, `-G`, `-g`, `-p`, `-T`, `-t`, `-U`, and `-u` options. If more than one of these options are given, then `ps` will select all processes which are matched by at least one of the given options.

For the processes which have been selected for display, `ps` will usually display one line per process. The `-M` option may result in multiple output lines (one line per thread) for some processes. By default all of these output lines are sorted first by controlling terminal, then by process ID. The `-m`, `-r`, and `-v` options will change the sort order. If more than one sorting option was given, then the selected processes will be sorted by the last sorting option which was specified.

For the processes which have been selected for display, the information to display is selected based on a set of keywords (see the `-L`, `-O`, and `-o` options). The default output format includes, for each process, the process ID, controlling terminal, CPU time (including both user and system time), state, and associated command.

```
$ ps -o pid,ppid,pgid,sid,comm
ps: sid: keyword not found
  PID  PPID  PGID COMM
  251   250   251 -bash
  256   255   256 -bash
  260   259   260 -bash
  837   260   837 man
  838   837   837 sh
  839   838   837 sh
  843   839   837 sh
  845   843   837 /usr/bin/less
  266   265   266 -bash
  791   266   791 vi
  275   274   275 -bash
  279   278   279 -bash
```

A process belongs to a process group, and a process group belongs to a session. The session may or may not have a controlling terminal. If it does, then the terminal device knows the process group ID of the foreground process. The process group ID of the foreground process can be set in the terminal driver with the `tcsetpgrp()` function. The foreground process group ID is an attribute of the terminal, not of the process. This value is what ps prints as TPGID on some systems (Linux).

The following sequence of commands illustrates how the (bash) shell handles pipelines:

```
$ ps -ao pid,ppid,pgid,comm | cat
  PID  PPID  PGID COMM
 1062   162  1062 login
 1063  1062  1063 -bash
 1083  1063  1083 ps
 1084  1063  1083 cat
```

The `-a` option forces the list to include the `ps` command itself. The last two processes in the pipeline (in the list) are the children of the shell, and the shell is the child of `login`. The shell forks and each of the two processes are exec-ed by copies of the shell. Also, notice that the process group IDs are both set to the leader process, which is `ps`.

If the same pipeline is executed in the background:

```
$ ps -ao pid,ppid,pgid,comm | cat &
[1] 1088
$   PID  PPID  PGID COMM
 1062   162  1062 login
 1063  1062  1063 -bash
 1087  1063  1087 ps
 1088  1063  1087 cat

[1]+  Done                    ps -ao pid,ppid,pgid,comm | cat
```

…only the process IDs of `ps` and `cat` change, and not their parent process IDs. The process group IDs of both `ps` and `cat` are now changed to the new process ID of `ps`.

If a background process tries to read from the terminal:

```
$ vi temp.txt &
[1] 1104
$ fg
vi temp.txt
```

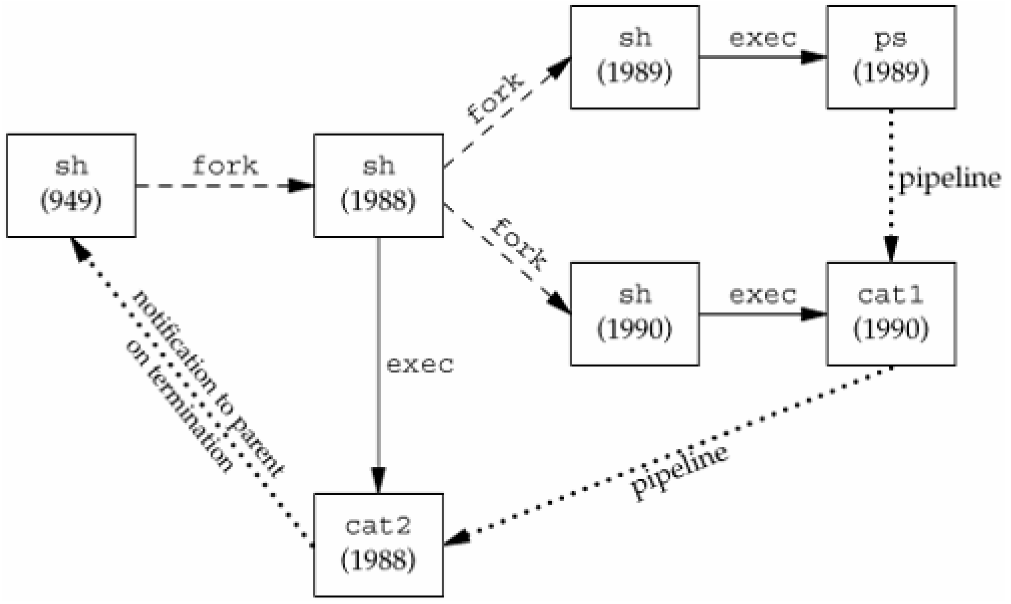
…the background job is placed into a background process group, which causes the signal SIGTTIN to be generated if the background job tries to read from the controlling terminal. Without job control, the shell redirects the stdin for the background process to `/dev/null`, which reads an EOF character and would terminate the input command.

If a background process, instead of reading from stdin, explicitly opens `/dev/tty`, may cause the prompt to send the desired input directly to the shell, causing it to try to interpret the input as a shell command, while the resulting error shell output line will be sent as input to the process. This means that both the shell and the bg process are attempting to read from the same device (the terminal) at the same time, and the result depends on the system.

If we execute three processes in a pipeline:

```
$ ps -ao pid,ppid,pgid,comm | cat | cat
  PID  PPID  PGID COMM
 1062   162  1062 login
 1063  1062  1063 -bash
 1109  1063  1109 ps
 1110  1063  1109 cat
 1111  1063  1109 cat
```

The shell forked three copies of itself, which each exec-ed one of the three commands, `ps`, and the 2 `cat`s. This reflects in the fact that the parent process IDs of these processes are the process ID of the shell. Also, the leader process, `ps`, has lent its process ID as the process group ID for the three processes in its group.



Orphaned Process Groups

A process whose parent terminates is called an **orphan**, and is inherited by `init`. Entire process groups can become orphaned and inherited by `init`.

This program example forks, creating a parent and a child. The parent sleeps for 5 seconds, allowing the child to execute and stop before the parent terminates. The child will use a signal handler (the function `sigHUP()`) for the SIGHUP signal. This function visualizes the fact that the child receives the SIGHUP signal.

The child stops, by sending itself the SIGTSTP signal with the `kill()` function. This suspends the child (just like a Ctrl-Z key combination). The child becomes orphaned, its parent process ID becomes 1.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

void PrintIDs(char *name);
void SigHUP(int signo);

int main(void)
{
     char c;
     pid_t pid;

     PrintIDs("Parent: ");
     pid=fork();
     if (pid>0) /* parent */
     {
          sleep(5); /* sleep to let child stop */
          exit(0);
     }
     else /* child */
     {
          PrintIDs("Child: ");
          signal(SIGHUP, SigHUP); /* SigHUP() becomes the signal
handler */
          kill(getpid(), SIGTSTP); /* stop child (self) */
          PrintIDs("Child: ");
          if (read(STDIN_FILENO, &c, 1)!=1)
          {
               printf("read() error from controlling TTY, errno=
%d\n", errno);
               printf("%s\n", strerror(errno));
          }
          exit(0);
     }
}
void PrintIDs(char *name)
{
     printf("%s: pid=%d, ppid=%d, pgrp=%d, tpgrp=%d\n",
          name, getpid(), getppid(), getpgrp(),
tcgetpgrp(STDIN_FILENO));
     fflush(stdout);
}

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

The output is:

```
$ ./myorphan
Parent: : pid=1238, ppid=1063, pgrp=1238, tpgrp=1238
Child: : pid=1239, ppid=1238, pgrp=1238, tpgrp=1238
SIGHUP received, pid=1239
Child: : pid=1239, ppid=1, pgrp=1238, tpgrp=1063
read() error from controlling TTY, errno=5
Input/output error
```