Process Relationships and Job Control

- Process Groups
- Process Group API
- Creating a Process Group
- Sessions
- Session Controlling Terminal
- `setsid()` Function
- Job Control Basics
- Job Control Examples
- Background Job Input
- Background Job Output
- Shell Execution Without Job Control
- Shell Execution With Job Control
- Background Job Without Job Control
- Background Shell Execution With Job Control
- PipeLines Without Job Control
- Pipeline Execution With Job Control
- Background Pipeline Without Job Control
- Background Pipeline Execution With Job Control
- Orphaned Process Groups
- Orphaned Group Example
- Orphaned Group Example Continued

- Orphaned Group Example Logs
- References
- References

Process Groups

- Each process belongs to a process group.
- Each process group has a *process-group ID* PGID.
- A process with PID == PGID is the *process group leader*.
- A process group can continue to exist even though its process group leader may have terminated.
- Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input: processes that have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read.
- A job-control shell allows user to interact with jobs. Interface using & suffix, `fg`, `bg`, `jobs` using shell-assigned job id’s (not process id’s).

Process Group API

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

- `getpgrp()` returns the process group ID of the current process. Group leader if it matches the pid.

- `setpgid()` sets the process group ID of the process specified by `pid` to `pgid`. If `pid` is zero, the process ID of the current process is used. If `pgid` is zero, the process ID of the process specified by `pid` is used.

- A process can set the `pgid` only of itself or one of its children (provided it hasn't `exec()`'d).

Creating a Process Group

When a shell creates a pipeline, it wants to put them into a process group for easy control via job-control signals.

- If it tries to change the process group of its children, then it must do so before the child does an `exec()` which constitutes a potential race condition.
- If the child sets the process group, then there is a race condition in ensuring that is done before the parent sends any signals.
- Solution is to have both parent and child set the process group.

Sessions

- A session is a collection of process groups.
- Typical situation:

```
$ prog1 | prog2 &
$ prog3 | prog4 | prog5 &
$
```

The login shell is the session leader of a session which contains the two pipelined process groups.

Session Controlling Terminal

- A session can have upto 1 *controlling terminal*.

- Process that establishes the connection to the controlling terminal is called the *session leader*.

- If a session has a controlling terminal, then there is a single *foreground* process group and 0 or more *background* process groups.

- Only foreground process group can read from terminal and receive signals generated from a terminal like `SIGINT` (`control-C`), `SIGQUIT` (`control-\`), `SIGSTP` (`control-Z`), etc.

- Interrupt [Quit] key (typically `^C` [^\]) sends SIGINT [SIGQUIT] to all processes in the foreground process group.

- Terminal disconnect sends a SIGHUP to the session leader.

`setsid()` Function

```
pid_t setsid(void);
```

- Calling process must not be a process group leader.
- Process becomes session leader of a new session.
- Process becomes group leader of a new group.
- Process has no controlling terminal.

To ensure that controlling process is not a process group leader, parent calls `fork()` and terminates, while the child does a `setsid()`.

Job Control Basics

Job control refers to the ability to selectively stop (*suspend*) the execution of processes and continue (*resume*) their execution at a later point.

Requires:

- Support from shell (`csh`, `ksh`, `bash` support job control; classical `sh` does not).
- Support from the terminal driver (`^Z` generates `SIGTSTP`).
- Job control signals (`SIGSTOP`, `SIGCONT`).

Job Control Examples

- Start job in foreground

```
$ emacs main.c
```

- Start jobs in background.

```
$ ghostview &
[2] 26789
$ xterm &
[3] 26791
$
[2]-  Done                    ghostview
[3]+  Done                    xterm
$ jobs
[1]+  Running                 emacs &
$
```

Background Job Input

```
$ cat > ~/tmp/t &
[2] 26798
$

[2]+  Stopped (tty input)     cat >~/tmp/t
$ fg
cat >~/tmp/t
hello world
^D
$ cat ~/tmp/t
hello world
$
```

Background Job Output

```
$ cat ~/tmp/t &
[2] 26802
$ hello world

[2]+  Done                    cat ~/tmp/t
$ stty tostop
$ cat ~/tmp/t &
[2] 26804
$

[2]+  Stopped (tty output)    cat ~/tmp/t
$ fg
cat ~/tmp/t
hello world
$
```

Shell Execution Without Job Control

No job control for non-interactive shells. Hence run shell-script to simulate no job control.

```
$ cat t
ps -o user,pid,ppid,pgid,tpgid,sid,comm
$ ./t
USER        PID  PPID  PGID TPGID   SID COMMAND
umrigar  14768 14767 14768 26927 14767 bash
umrigar  26927 14768 26927 26927 14767 bash
umrigar  26928 26927 26927 26927 14767 ps
$
```

ps `26928` is in same process group `26927` as shell `26927` which launched it.

Shell Execution With Job Control

```
$ ps -o user,pid,ppid,pgid,tpgid,sid,comm
USER       PID  PPID  PGID TPGID   SID COMMAND
umrigar  14768 14767 14768 26918 14767 bash
umrigar  26918 14768 26918 26918 14767 ps
```

ps 26918 is in its own process group compared to shell 14768 which launched it.

Background Job Without Job Control

```
$ cat t
ps -o user,pid,ppid,pgid,tpgid,sid,comm &
$ ./t
```

with output:

```
USER        PID  PPID  PGID TPGID   SID COMMAND
umrigar  14768 14767 14768 14768 14767 bash
umrigar  26931     1 26930 14768 14767 ps
```

Intermediate shell with presumed PID 26930 (which simulates non-job-control shell) has terminated. Note that ps is in same process group 26930 as shell which launched it. Since command is run in background, note that TPGID associated with login shell.

Background Shell Execution With Job Control

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

with output:

```
  USER         PID  PPID  PGID TPGID   SID COMMAND
  umrigar  14768 14767 14768 14768 14767 bash
  umrigar  26919 14768 26919 14768 14767 ps
```

Again, `ps` is in its own process group.

PipeLines Without Job Control

```
$ cat t
ps -o user,pid,ppid,pgid,tpgid,sid,comm | cat
$ ./t
USER        PID  PPID  PGID TPGID   SID COMMAND
umrigar  14768 14767 14768 26934 14767 bash
umrigar  26934 14768 26934 26934 14767 bash
umrigar  26935 26934 26934 26934 14767 ps
umrigar  26936 26934 26934 26934 14767 cat
$
```

Command-line shell `14768` launched intermediate shell `26934` (which simulates non-job-control shell) which launched pipeline `ps` `26935` and `cat` `26936`. Once again, pipeline is in same process group `26934` as shell which starts it.

Pipeline Execution With Job Control

```
$ ps -o user,pid,ppid,pgid,tpgid,sid,comm | cat
USER       PID  PPID  PGID TPGID   SID COMMAND
umrigar  14768 14767 14768 26920 14767 bash
umrigar  26920 14768 26920 26920 14767 ps
umrigar  26921 14768 26920 26920 14767 cat
```

Pipeline is in its own process group `26920` which is different from the process group `14768` of shell which started it.

Background Pipeline Without Job Control

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

with output:

```
  USER        PID  PPID  PGID TPGID   SID COMMAND
  umrigar  14768 14767 14768 14768 14767 bash
  umrigar  26939     1 26938 14768 14767 ps
  umrigar  26940     1 26938 14768 14767 cat
```

Intermediate shell has terminated. Note that `TPGID` corresponds to command-line shell. Pipeline is in same process group `26938` as presumed PID `26938` of intermediate shell.

Background Pipeline Execution With Job Control

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

with output:

```
USER        PID  PPID  PGID TPGID   SID COMMAND
umrigar  14768 14767 14768 14768 14767 bash
umrigar  26922 14768 26922 14768 14767 ps
umrigar  26923 14768 26922 14768 14767 cat
```

Pipeline is in its own process group 26922.

Orphaned Process Groups

- An orphaned process group is one which has no process with a parent in a different process group but in the same session.
- If a process group is orphaned, then there is no parent which can restart a stopped process in the process group.
- Every process in a newly orphaned group is sent SIGHUP followed by SIGCONT.
- If a orphaned process group attempts terminal I/O, it gets an error (if it was stopped, there is no possible way it can be restarted since there is no process to send it a continue signal).

Orphaned Group Example

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

  outIDs("parent");
  if ((pid = fork()) < 0) {
    perror("fork error"); exit(1);
  }
  else if (pid > 0) { /* parent */
    sleep(5);
    exit(0);
  }
  else { /* child */
    char c;
    outIDs("child");
    signal(SIGHUP, sighup);
    kill(getpid(), SIGTSTP);
    outIDs("child");
    if (read(0, &c, 1) != 1) {
      perror("read error"); exit(0);
    }
  }
  return 0;
}
```

Orphaned Group Example Continued

```
static void
sighup(int signo)
{
  printf("SIGHUP: pid = %d\n", (int)getpid());
  return;
}

static void
outIDs(const char *name)
{
  printf("%s: pid = %d, ppid = %d, pgrp = %d\n",
         name, (int)getpid(), (int)getppid(),
         (int)getpgrp());
  fflush(stdout);
}
```

Orphaned Group Example Logs

```
$ ./orphgrp
parent: pid = 26986, ppid = 26985, pgrp = 26986
child: pid = 26987, ppid = 26986, pgrp = 26986
$ SIGHUP: pid = 26987
child: pid = 26987, ppid = 1, pgrp = 26986
read error: Input/output error

$
```

References

References

Text, Ch. 34.

APUE, Ch. 9, 34.

Jim Frost, *UNIX Signals and Process Groups*, at http://www.cis.temple.edu/~ingargio/old/cis307s96/readings/docs/signals.html.

FSF, *The GNU C Library*, at http://www.gnu.org/software/libc/manual/. Specifically, Implementing a Job Control Shell at http://www.gnu.org/software/libc/manual/html_node/Implementing-a-Shell.html.