Processes and Job Control

Hour 17

- Objectives
  - Definitions: process, orphan, and zombie
  - System processes
  - Process creation
  - Examining processes: the ps command
  - Job control: &, nohup, fg, bg, jobs, ( ), and kill
  - Exit status

Process

Also called a job by C and Korn shells

- When a program or executable file is loaded from disk and started running (i.e., when a command is run), it is called a process
  - identified by a unique process ID (PID) number
  - has an owner
  - private data

- A program can be loaded more than once
  - creates multiple processes
  - each process has a different PID
  - each process may have a different owner

- PIDs are unique, nonnegative integers
  - numbers recycle without collisions
System Processes
Processes created during system boot

- **0** System kernel
  - "hand crafted" at boot
  - called **swap** in older versions (swaps the CPU between processes)
  - called **sched** in newer versions (schedules processes)
  - creates process 1
- **1** **init** (the parent of all processes except process 0)
  - general process spawner
  - begins building locale-related environment
  - sets or changes the system run-level
- **2** page daemon (**pageout** on most systems)
- **3** file system flusher (**fsflush**)

Process Life Cycle
Overview of creating new processes

- **fork** creates two identical processes (parent and child)
- **exec**
  - replaces the process’s instructions or program with the instructions for another program
  - because it is the same process, it does not change the PID
- The system maintains completed processes in a quiescent state until their exit status is examined by their parent
Zombies and Orphans

Special process states

- A **zombie** is a process that has exited but whose parent hasn’t yet checked its exit status
  - Waiting for a parent to wait on it
  - Retains PID, termination status, and CPU time
  - Called a zombie in BSD days
  - Called <defunct> in SVR4
- An **orphan** is a process whose parent dies before it does
  - Orphans are adopted by init
  - Some systems kill background processes at logout; the `nohup` utility sees that they are adopted by init instead

Background Processes

Concurrent batch processing

- When a shell interprets a command, it loads a program and starts a process
  - it displays the prompt only when the command (process) finishes
  - the command gets standard in; standard out & error go to the screen
- `command &`
  - runs `command` (process) but shell doesn’t wait (prints prompt and accepts new commands immediately)
  - `command` may be simple or may be a pipe
  - background processes often redirect output
- `nohup command &`
  - makes command immune to SIGHUP and SIGTERM (often sent at logout)
  - background process continues to run after logout
  - if command output is not explicitly redirected, the shell redirects output to `nohup.out`
The **ps** Command

Obtaining process information

- **ps [ options ]**
  - `-l` long listing (much additional information)
  - `-a` all (include processes owned by other users)
  - `-x` (BSD) include daemons, system, and background processes
  - `-w` (BSD) wide listing (full name of command)
  - `-e` (SVR4) list information about every process (like BSD `-ax`)

**Output**

- **UID** user ID (process owner)
- **PID** process ID
- **PPID** parent process ID
- **PRI** priority
- **NI** nice
- **TTY** starting terminal
- **CMD** the command
- **S** status

---

**ps** Example

BSD first and SVR4 output

```plaintext
% ps -axl
F   UID   PID  PPID %C PRI NI   SZ  RSS    WCHAN S TT        TIME COMMAND
19     0     0     0  0  96  0    0    0          T ?         0:00 sched
8     0     1     0  55 20 656 136 iwscn_di S ?        16:47 /etc/init -
19     0     2     0  98  0    0    0 push_cv S ?        0:05 pageout
19     0     3     0  60  0    0    0 fsflush_ S ?        279:34 fsflush
8     0 28184 28180  0   0       0    0                   Z  0:00 <defunct>
8     0 29550 515  0  0  55 20 2772 1892 nksyms_i S ?    0:00 /usr/openwin
8     0 29556 1  0  57 20 3044 1868 pm_state S ?         0:07 xconsole -ge
8     0 4923 4921  0  0  55 20 1628 1264 nksyms_c S pts/3 0:00 login -p -d
8 1288 4924 4923  0  58 20 1052 852 nksyms_c S pts/3 0:00 -csh
8     0 5023 4924  0  0  0    0    0                   O pts/3 0:00 ps -axl

% /bin/ps -el
F S   UID   PID  PPID  C PRI NI ADDR    SZ  WCHAN TTY      TIME CMD
19 T   0 0 0 0 0 SY f0274e38     0  ?        0:01 sched
8 S   0 1 0 0 0 44 20 f5b27888 164  f5b27a80 ?        16:48 init
19 S   0 2 0 0 0 SY f5b271c8 0 f02886a4 ?        0:06 pageout
19 S   0 3 0 1 0 SY f5b26b08 0 f028aeb4 ?        279:35 fsflush
8 S  0 17012 279 0 0                   0:00 <defunct>
8 O   0 5024 4924  1 61 20 f65128f8 201 pts/3 0:00 ps
8 S   0 19964 515  0 40 20 f68dcbe0 693  f62432ae ?          0:00 xdm
8 S   0 506 515  0 41 20 f655f248 695  f6482756 ?          0:02 xdm
8 S   0 4923 4921  0 71 20 f6ce4e4d0 407  f6c0e540 pts/3 0:00 login
8 S 1288 4924 4923  0 41 20 f6bc96d0 263  f6bc98c8 pts/3 0:00 csh
```
Job Control

Implemented in the shell

- **jobs [-l]** lists current background processes
  - `-l` adds PIDs to output
- **bg [%job]** puts job in background
  - last referenced job if no job is specified
- **fg [%job]** restarts job in foreground
  - last referenced job if no job is specified
- **control-z** suspend or stop a foreground job
- **%job**
  - `%n` where n is the job number
  - `%prefix` command name or first characters of the command name
  - `%+` the last job referenced
  - `%–` the second to the last job referenced

---

Signaling Processes

Simple interprocess communication

- **kill [-l] [-signal] pid ...**
  - signal may be a signal number or signal symbol
- **kill [-1] [-signal] %job**
  - %job as defined on the previous slide
- **Signals are only sent from process to another**
  - kill causes the shell to send a signal to the indicated process
  - kill is a general signal sender; it is so named because the kill (9 or SIGKILL) signal is one of the most commonly sent signals
- **To find signal numbers and symbols**
  - See /usr/include/sys/signal.h
  - use the -l option
Process Termination

- **exit functions and return operator**

- **exit status**
  - # shell script

- **C**
  - `void exit(int status); /* ANSI */`
  - `void _exit(int status); /* POSIX */`
  - `return status;`
    - Terminates a program only when called from main
    - equivalent to calling `exit`

- **status**
  - 0  normal, non-error termination
  - 1-255 abnormal, error termination

- In Bourne and Kom, the variable `$?` represents the exit status

Sub-Shell Processes

- Implemented in the shell

- `(command)`
  - the shell spawns a sub-shell to execute commands placed within parentheses
  - `command` may be a simple command, `command` with options and arguments, a pipe, or a sequence of commands
  - the parent and sub-shell execute concurrently

- **Example**
  - `tar cf - . | (cd /backup; tar xf -)`
    - the first tar command is executed by the parent shell
    - the cd and second tar command are executed by the child shell
    - the first tar command creates an archive and writes it standard out
    - the second tar command reads standard out and extracts the archive