Processes and Job Control

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

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
- 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
- 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

* BSD and SVR4 versions have different options and output
* `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
  * `VIRT` virtual memory
  * `RES` resident memory
  * `SHR` shared memory
  * `STK` stack
  * `TIME` CPU time
  * `MEM` memory
  * `COMMAND` command name

Example

```
% ps axl
```

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
  * `%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

Signalizing Processes

Simple interprocess communication

* `kill [-l] [-signal] [ pid ... ]`
  * signal may be a signal number or signal symbol
  * `%signal` 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
  * %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`
    * `[ ... ] ( 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