Process Control

Process

- Abstraction for a running program
  - Manages program's use of memory, CPU time, and I/O resources
- Most of the work is done in the context of the process rather than handled separately by the kernel

Components of a process

- Address space
  - Set of pages allocated to the process by the kernel
  - Code and libraries being executed by the process (code segment)
  - Variable space (data segment)
  - Stack
- A set of data structures within the kernel
  - Process address space map
  - Current state of the process (R, S, T, Z)
  - Execution priority
  - Resources used by the process
  - Signal mask to know the signals to be blocked
  - Owner of the process
- Process may have information about each execution context, or thread
  - Scheduling is on process level and not thread level
  - Threads have little impact on system administration at present
- Common parameters of a process from system administration viewpoint
  - PID
    * Assigned by the kernel
    * Unique for every process
    * Most important information to identify any process
  - PPID
  - UID and EUID
    * UID is initialized from the EUID of the parent
  - GID and EGID
  - Nice value
  - Control terminal
    * Determines default linkages for stdin, stdout, and stderr

Life cycle of a process

- Created by `fork(2)`
- Program text changed by one of the calls from the exec family
- The grand-ancestor of every process is `init` with PID as 1
- Process termination
  - Process terminates by calling `_exit(2)`
  - It supplies an integer to `_exit(2)` to identify the reason for termination; 0 (or `EXIT_SUCCESS`) for successful termination
  - Process death must be acknowledged by its parent, by using a call to `wait(2)`
  - Parent picks up the reason for termination and a summary of the child's use of resources
  - If child outlives its parent, it is adopted by `init` who performs the last rites upon child's death

Signals

- Process-level interrupt requests
  - Used for communication among processes
  - Used to kill, interrupt, or suspend a process from terminal driver
  - Sent by administrator to achieve different tasks
  - Sent by kernel to issue a trap, or to report synchronous errors
- Signal is handled by a designated handler function, or the default handler provided by kernel
  - Handler *catches* the signal
  - After catching the signal, execution resumes from the point where signal is caught
- Signals can be ignored or blocked by a process
  - Ignored signal is simply discarded
  - Blocked signal is queued for delivery till the time the process unblocks the signal
  - The handler for a newly unblocked signal is called only once even if there are several signals waiting when it is unblocked
- Signals of interest to system administrators on Solaris (complete list available in `/usr/include/sys/signal.h`)

#	Name	Description	Default	Catch	Block	Dump core
1	`HUP`	Hangup	Terminate	Y	Y	N
2	`INT`	Interrupt (rubout)	Terminate	Y	Y	N
3	`QUIT`	Quit	Terminate	Y	Y	Y
9	`KILL`	Kill	Terminate	N	N	N
10	`BUS`	Bus error	Terminate	Y	Y	Y
11	`SEGV`	Segmentation violation	Terminate	Y	Y	Y
15	`TERM`	Software termination signal from kill	Terminate	Y	Y	N
16	`USR1`	User defined signal 1	Terminate	Y	Y	N
17	`USR2`	User defined signal 2	Terminate	Y	Y	N
20	`WINCH`	Window size change	Ignore	Y	Y	N
23	`STOP`	Stop	Stop	N	N	N
24	`TSTP`	User stop requested from tty	Stop	Y	Y	N
25	`CONT`	Stopped process continued	Ignore	Y	Y	N

  - Signal names are prefixed with `SIG` in the file

  - `HUP` provides a reset request for daemons, or if possible, makes the daemon to read the configuration file again without restarting
    * It is used to clean up or kill the process associated with the tty when the user logs out
    * C shell family shells make the background processes immune to `HUP`
    * Bourne shell family shells need to emulate the behavior with `nohup` command
  - `INT` is sent by terminal driver in response to `^C`
    * Simple programs should quit (if signal is caught), or allow themselves to be killed (if signal is not caught)
    * Programs waiting for user input should stop what they are doing, clean up, and wait for user input again
  - `QUIT` is similar to `TERM` but produces a core dump if not caught
  - `KILL` terminates a process at OS level and is not receivable by the process
  - `TERM` is a request to terminate execution completely

kill to send signals

Process states

- Five execution states for a process in Unix

  O Running Process is running on a processor

  S Sleeping Process is waiting for an event to complete

  R Runnable Process is on run queue; ready to run

  Z Zombie Process terminated and parent not waiting

  T Trace Process is stopped, either by a job control signal or because it is being traced

- Stopped processes are administratively forbidden to run
  - Processes are stopped by `STOP` or `TSTP` signals
  - Restarted with `CONT`
  - Stopped is similar to sleeping except that the process has to be explicitly woken up (or killed) by some other process

nice and renice to influence scheduling priority

- Numeric hint to kernel about process priority
  - Higher nice value implies lower priority
  - Negative nice value implies higher priority
  - Higher the number, lower the priority
- Priority assigned to a process can be changed by changing its *nice value*, using the command `nice`
- The processes can only be made *more nice* (decrease priority) by users, superuser can make the processes *less nice* (increase priority)
- Each process is assigned a default nice value of 20
- Example

```
nice -10 sort foo
```

will execute the command at a lower priority

- Most common nice value range is from −20 to +19
- Newly created process inherits the nice value from parent
- The kernel may increase the nice value for process that use excessive CPU time
- Confusion due to shell-built-in nice and system nice commands

ps to monitor processes

- SYS V vs BSD versions
- Without options, the `ps` command lists the processes of the user who invoked the command

```
$ ps
   PID TTY      TIME CMD
 29677 pts/3    0:00 xbiff
 29676 pts/3    0:04 xdaliclo
 29739 pts/3    0:01 vi
 29681 pts/3    0:02 xdvi.bin
 29742 pts/3    0:00 tcsh
 29659 pts/3    0:01 tcsh
$
```

Here, the command displays the process id, the controlling terminal, the CPU time used, and the name of the command or program

- We can get a more detailed listing of the processes by using the `-l` option (for *long*) with the `ps` command

```
$ ps -l
 F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME CMD
 8 R   122   318   315  1  81 20 f60b2488    459          pts/3    0:00 tcsh
 8 S   122    51    49  0  51 20 f625f720    522 f625f918 pts/3    0:02 tcsh
 8 S   122    93    51  0  40 20 f61da8d8    708 f5d04b26 pts/3    0:00 xbiff
 8 S   122    92    51  0  40 20 f5fc51f0    624 f5f80836 pts/3    0:22 xdaliclo
 8 S   122   315    51  0  61 20 f625fde0    367 f5e10738 pts/3    0:00 vi
 8 S   122   160    51  0  41 20 f5fc36f0    903 f5f805b6 pts/3    0:02 xdvi.bin
$
```

The fields can be described as follows

Field	Description
`F`	Associated flag; for historical reasons only; no special meaning
`S`	Running state of the process (as defined previously)
`UID`	User identifier
`PID`	Process identifier
`PPID`	Parent process identifier
`C`	Processor utilization for scheduling (obsolete)
`PRI`	Priority of the process
`NI`	Nice value (for scheduling)
`ADDR`	Address in memory
`SZ`	Data and stack segment size combined (in Kbytes)
`WCHAN`	Address of an event for which the process is waiting
`TTY`	Controlling terminal
`TIME`	Cumulative execution time for the process
`CMD`	Command being executed (up to first 80 characters)

- Another useful option with `ps` is `-e` which lists all the processes currently active on the system

Monitoring processes with top

- Regularly updated summary of active processes and their use of resources

Runaway processes

- Processes that use up too much of CPU resources
- Could be legitimate, or buggy, or malicious (like password cracker)
- If problem occurs in `/tmp` and it is a filesystem by itself, the partition can be reinitialized by the `newfs` command

Job control with background and foreground processes

- A *job* is a process that is either running in the background, or is stopped
  - Processes in the background continue to run but do not make the shell to wait for their termination before putting the prompt
  - You can have many processes running in the background at the same time
- When you issue a command on the shell, you cannot do any further work until it terminates
  - You can abort it by sending it a signal by pressing `^C`
- Job control is used to control multiple processes
- Allows placing the jobs into foreground or background, and move them from foreground to background or vice versa
- You can suspend the jobs temporarily, and restart them later
  - A running command can be *suspended* by pressing `^Z` key
  - After the command is suspended, user is presented with a new shell prompt
  - Suspended commands can be later resumed at the point where they were suspended
    - The command is resumed in foreground by typing `fg` on shell prompt

    * The command is resumed in background by typing `bg` on shell prompt

- Checking on the jobs associated with the terminal session
  - Use the command `jobs`
- Job control commands are summarized as:

Command	Action
`&`	Run the preceding command in background
`CTRL-Z`	Suspend a foreground job
`bg`	Run a suspended job in the background
`fg`	Run the command in foreground (from suspended or background)
`jobs`	List active and suspended jobs in the background
`kill`	Terminate a job
`stop`	Suspend a background job (not in `ksh`)
`suspend`	Equivalent of `CTRL-Z` in `ksh`