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
Process Control

- 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`)

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

- 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 \(^\text{C}\)
  - 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
Process Control

```
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  f61da9d8 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 f5f805b6 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
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Associated flag; for historical reasons only; no special meaning</td>
</tr>
<tr>
<td>S</td>
<td>Running state of the process (as defined previously)</td>
</tr>
<tr>
<td>UID</td>
<td>User identifier</td>
</tr>
<tr>
<td>PID</td>
<td>Process identifier</td>
</tr>
<tr>
<td>PPID</td>
<td>Parent process identifier</td>
</tr>
<tr>
<td>C</td>
<td>Processor utilization for scheduling (obsolete)</td>
</tr>
<tr>
<td>PRI</td>
<td>Priority of the process</td>
</tr>
<tr>
<td>NI</td>
<td>Nice value (for scheduling)</td>
</tr>
<tr>
<td>ADDR</td>
<td>Address in memory</td>
</tr>
<tr>
<td>SZ</td>
<td>Data and stack segment size combined (in Kbytes)</td>
</tr>
<tr>
<td>WCHAN</td>
<td>Address of an event for which the process is waiting</td>
</tr>
<tr>
<td>TTY</td>
<td>Controlling terminal</td>
</tr>
<tr>
<td>TIME</td>
<td>Cumulative execution time for the process</td>
</tr>
<tr>
<td>CMD</td>
<td>Command being executed (up to first 80 characters)</td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Run the preceding command in background</td>
</tr>
<tr>
<td>CTRL-Z</td>
<td>Suspend a foreground job</td>
</tr>
<tr>
<td>bg</td>
<td>Run a suspended job in the background</td>
</tr>
<tr>
<td>fg</td>
<td>Run the command in foreground (from suspended or background)</td>
</tr>
<tr>
<td>jobs</td>
<td>List active and suspended jobs in the background</td>
</tr>
<tr>
<td>kill</td>
<td>Terminate a job</td>
</tr>
<tr>
<td>stop</td>
<td>Suspend a background job (not in <code>ksh</code>)</td>
</tr>
<tr>
<td>suspend</td>
<td>Equivalent of CTRL-Z in <code>ksh</code></td>
</tr>
</tbody>
</table>