Process Control

top: The top command displays processor activity of your Linux box and also displays tasks managed by the kernel in real-time. It shows processor and memory usage and other information like running processes. It is found in UNIX-like operating systems.

You can use top with the u flag to show only processes for a specific user:

```
top -u blake
```

You can press 'z' with top running to color-code running processes.

Other key switches:
h - Show the current version
c - This toggles the command column between showing command and program name
d - Specify the delay time between refreshing the screen
o - Sorts by the named field
p - Only show processes with specified process IDs
u - Show only processes by the specified user
Shift+p - order by CPU usage
q - quit

You can kill a process while running top by pressing 'k' and then supplying its PID (process ID).

ps: By default, the ps command produces a list of all processes associated with the current user and terminal session.

The PID is the process ID which identifies the running process. The TTY is the terminal type.

Use `ps -e` to see all processes. The output can be quite large, so this is often used in conjunction with grep, more, or less.

Use `ps r` to see only running processes.
`ps -p <pid>` looks for a specific process by ID
You can also do `ps -C <command>` For example, to see if Chrome is running:
`ps -C chrome`

`ps aux` gives us a more complete and detailed picture of the processes.

`ps axjf` shows us a tree view that illustrates the hierarchical relationship between parent and child processes. A process's parent is the process that was responsible for spawning it. If a process's parent is killed, then the child processes also die. The parent process's PID is referred to as the PPID.

pgrep: The `pgrep` command is a quick way of getting the PID of a process. For example:

```
pgrep bash
```

outputs 6163 on my system currently.
There will be no output if the input is not associated with an actual process.

kill: All processes in Linux respond to signals. Signals are an operating system level way of telling programs to terminate or modify their behavior. The most common way of passing signals to a program is with the `kill` command. As you might expect, the default functionality of this utility is to attempt to kill a process. To kill gedit, for example:

```
pgrep gedit
6270
kill 6270
```

This sends the TERM signal to the process. The TERM signal tells the process to please terminate. This allows the program to perform clean-up operations and exit smoothly.

If the program is misbehaving and does not exit when given the TERM signal, we can escalate the signal by passing the KILL signal with:

```
kill –KILL
```

or

```
kill -9
```

This signals the operating system kernel to shut down the program. Try TERM first.

You can list all of the signals that are possible to send with kill by typing:

```
kill -l
```

pkill: The `pkill` command works in almost exactly the same way as `kill`, but it operates on a process name instead:

```
pkill -9 gedit
```

is the same as:

```
kill -9 `pgrep gedit`
```

killall: Use `killall` to kill every instance of a certain process:

```
killall firefox
```

nice: Some processes might be considered mission critical for your situation, while others may be executed whenever there are leftover resources. Linux controls priority through a value called niceness. High priority tasks are considered less nice, because they don't share resources as well. Low priority processes, on the other hand, are nice because they only take minimal resources.

When we ran top, there was a column marked "NI". This is the nice value of the process. These range from -20 to 20 or -19 to 19 depending on the system.

To run a program with a certain nice value, we can use the `nice` command:

```
nice -n 15 firefox
```

This only works when beginning a new program. To alter the nice value of a program that is already executing, we use a tool called `renice`:

```
renice 15 6440
```

or

```
renice 15 `pgrep firefox`
```

`nice` operates with a command name while `renice` operates with a PID.

Background Jobs

When you execute a shell-script or command that takes a long time, you can run it as a background job. This is useful, for example, in applications with GUIs. If I do:

```
firefox
```

Then I can't use my shell anymore while firefox is running because it will be busy making firefox happen in the foreground. If I do

```
firefox &
```

Then firefox runs as a job in the background and I can continue to use my shell. In general, appending an ampersand (&) to the command runs the job in the background. The shell does not wait for the command to finish, and the return status is 0. Note that background jobs will close if/when you close the shell that started them.

If we have a job running in the foreground that we want to send to the background, we can press CTRL+Z to suspend the current foreground job and then execute the command `bg` to make that job execute in background.

By default, `bg` resumes the shell's notion of the "current" suspended job in the background. This is probably the most recently suspended job. For more control, you can use `bg [job]` where `[job]` is the number of the job that you want to run in the background. Job number 1 is referred to as %1, job number 2 is referred to as %2, etc.

%, %+, or %% refers to the current job;
%- or - refers to the previous job.

You can list out the background jobs with the command `jobs`. This will show the jobs (column 3) with their statuses (column 2) and job IDs (column 1). It will also indicate which job is current and which is previous with a + and – respectively in column 1 after the ID.

You can bring a background job to the foreground with the `fg` command. When executed without arguments, it will take the current background job to the foreground. For more control, use a job ID the same way we did with `bg`.

You can kill a specific background job with `kill %job-number`. For example:
`kill %2`