COMP 2718: Processes

By: Dr. Andrew Vardy

Adapted from the notes of Dr. Rod Byrne

Outline

- Processes — Chapter 10 of TLCL
- What is a Process
- How a Process Starts
- `ps` - View Processes
- `top` - View Processes Dynamically
- Foreground and Background
- Job Control
- Stopping a Process
- Signals
- `kill` Doesn't Always Mean Kill!

Processes — Chapter 10 of TLCL

We're going to cover **processes** from chapter 10 of the textbook. Along the way, we'll introduce the following commands:

- `ps` – Report a snapshot of current processes
- `top` – Display tasks
- `jobs` – List active jobs
- `bg` – Place a job in the background
- `fg` – Place a job in the foreground
- `kill` – Send a signal to a process
- `killall` – Kill processes by name

What is a Process

A **process** is a running instance of a particular **program**. It is a useful word because the same program (e.g. `cat`) can have many different running instances. That is, many processes.

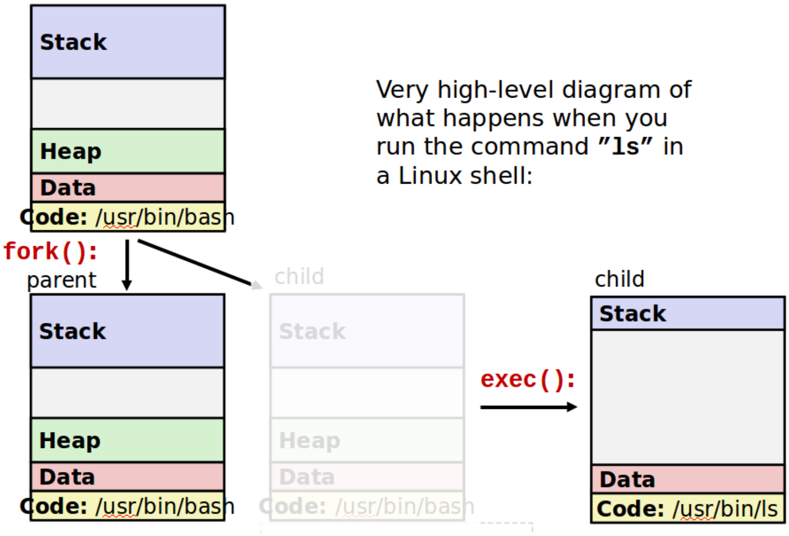
Modern OS's give the *appearance* of running many processes simultaneously even when limited to a single-core CPU. This is done by rapidly switching between process, so that all processes appear active to the user. This is known as **multitasking**.

How a Process Starts

One process is always launched from another process. The first is the **parent process** while the second is the **child process**. In Unix and Unix-like systems, a process is launched when a process **forks**. A fork is a system call (a request to the OS kernel) which creates a copy of the calling process. Another system call caled **exec** which replaces the code and memory of the parent process with those of the child process.

Why this strange strategy?

Interprocess communication: The child process will get a copy of the parent's **file descriptors** and therefore get input from the parent (e.g. through a pipe). Also, the parent will get an **exit status** from the child when it terminates.



Very high-level diagram of what happens when you run the command **"ls"** in a Linux shell:

On Linux, there is a program called `init` which the kernel launches.

- `init` launches a bunch of *init scripts* in `etc`
- Some are **daemon programs** which run various services

All processes have a **process ID** or **PID**. For `init` the PID is 0. All other processes get incrementally increasing PIDs.

ps - View Processes

```
ps [options]
```

Without arguments, ps shows the processes associated with the current terminal (referred to as TTY—which comes form teletype).

```
$ ps
```

```
  PID TTY          TIME CMD
 2304 pts/0    00:00:00 bash
 2390 pts/0    00:00:00 ps
```

TIME means the CPU time consumed

`ps` is typically used with **BSD syntax** which does not use can use a leading dash for options. The most common form:

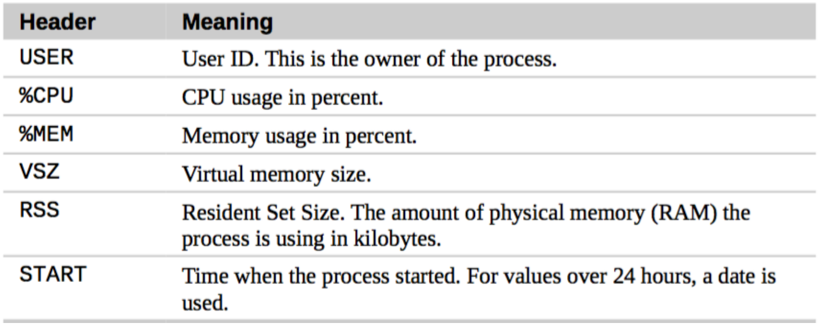
```
ps aux
```

a, u, and x mean:

a	List processes belonging to every user
u	Display in user-oriented format
x	List processes without a terminal (tty)

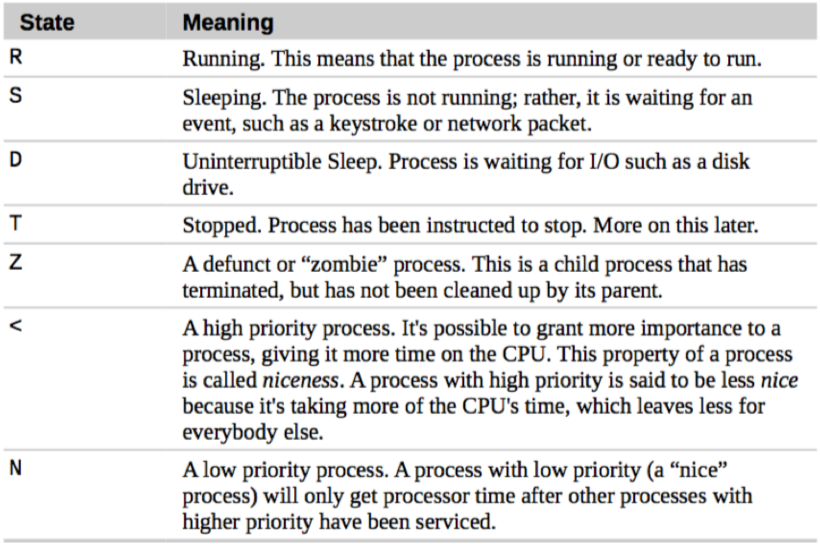
[Run examples]

What do these new columns mean ?



Header	Meaning
USER	User ID. This is the owner of the process.
%CPU	CPU usage in percent.
%MEM	Memory usage in percent.
VSZ	Virtual memory size.
RSS	Resident Set Size. The amount of physical memory (RAM) the process is using in kilobytes.
START	Time when the process started. For values over 24 hours, a date is used.

What does the STAT column mean?



State	Meaning
R	Running. This means that the process is running or ready to run.
S	Sleeping. The process is not running; rather, it is waiting for an event, such as a keystroke or network packet.
D	Uninterruptible Sleep. Process is waiting for I/O such as a disk drive.
T	Stopped. Process has been instructed to stop. More on this later.
Z	A defunct or “zombie” process. This is a child process that has terminated, but has not been cleaned up by its parent.
<	A high priority process. It's possible to grant more importance to a process, giving it more time on the CPU. This property of a process is called *niceness*. A process with high priority is said to be less *nice* because it's taking more of the CPU's time, which leaves less for everybody else.
N	A low priority process. A process with low priority (a “nice” process) will only get processor time after other processes with higher priority have been serviced.

`top` - View Processes Dynamically

`ps` is a fundamental command, but presents only a snapshot. By contrast `top` presents a continuous update (every 3 seconds) and highlights the *top* processes:

```
top
```

`top` displays some of the same info from `ps aux` but also other things including:

- Load average (upper right): Number of processes waiting to run (in a runnable state) averaged over 60 seconds, 5 minutes, and 15 minutes.
- "%Cpu(s)" (third line): Percentage CPU activity for user (us), system (sy), nice (ni), and idle (id) processes.

We will do some experiments using the `sysbench` tool for benchmarking CPU performance. This is probably not installed on your system. Under Linux install as follows:

```
sudo apt-get install sysbench
```

We will run the following to perform an extended CPU test:

```
$ sysbench --test=cpu --cpu-max-prime=10000000 run
```

Since this is hard to type, lets make an alias to it:

```
$ alias prime_test='sysbench --test=cpu \
> --cpu-max-prime=1000000 run'
```

Finally, lets run this in the **background** by ending the command with & (we will cover background processes shortly):

```
$ prime_test &
```

We can now run this several times to see the load on the system in `top`, which should be running in another terminal.

Foreground and Background

When the shell executes a program, that program is normally said to be running in the **foreground**. That is, we have to wait for it to exit before returning to the shell to do more work. Meanwhile, the foreground process has control of the terminal and can use the display in various ways (e.g. `vi`, `ninvaders`).

Since the OS supports multitasking, we can run multiple programs simultaneously. To run a program but immediately return control to the shell, we run the process in the **background**. This is done by suffixing the program with `&`.

```
$ gedit &
```

Any command can be run in the background, but a fast-running program like `ls` or `cat` would just return immediately anyway.

Try running in the foreground:

```
$ gedit
```

You can **interrupt a process** with “control-c”. This will send a signal to the process, asking it to quit (it may not respond). Note that the “control-c” must be entered at the terminal, so you may have to switch your window manager’s focus back to the terminal from `gedit`.

Job Control

The `jobs` command gives a listing of the jobs (i.e. processes) launched from the shell. e.g.

```
$ prime_test &
$ gedit &
$ jobs
[1]-  Running                         sysbench --test=cpu ...
[2]+  Running                         gedit &
```

You can refer to the above job numbers and bring a job to the foreground as follows:

```
$ fg %2  # Brings gedit to foreground
```

gedit can now be terminated with “control-C”.

Stopping a Process

A process in the foreground can be **stopped** (which really means "paused") with "control-z". For example, start `vi` then pause it with "control-z" to return to the shell.

```
$ vi
[In vi hit "control z"]
$ jobs
```

Now try running `gedit` and hitting "control-z" (at the terminal). Note how `gedit` has become unresponsive when the window size is adjusted. Thus. . .

A background process is disconnected from the terminal but is still running. A stopped process is not actively running.

We have introduced the following commands above:

Command	Description
`jobs` [args]	Display status of jobs
`fg` [jobspec]	Places the job in the foreground
`bg` [jobspec]	Places the job in the background

Notes:

- These commands refer to “jobs” not “processes”. A job may be a single process or a group of processes (e.g. a pipeline).
- A `jobspec` is a ‘%’ followed by the corresponding job number (from running `jobs`).

Signals

Processes can send each other signals. We have seen two already:

“control-c”	Sends a signal called INT (Interrupt) which generally means—please terminate.
“control-z”	Sends a signal called TSTP (Terminal Stop) which means—please stop.

Note that termination is different from stopping. A stopped process can be resumed. Also, both of the above signals can be ignored by the process.

There are many other types of signals. Here are some of the most common. . .

Number	Name	Meaning
1	HUP	Hangup. This is a vestige of the good old days when terminals were attached to remote computers with phone lines and modems. The signal is used to indicate to programs that the controlling terminal has “hung up.” The effect of this signal can be demonstrated by closing a terminal session. The foreground program running on the terminal will be sent the signal and will terminate.

2	`INT`	Interrupt. Performs the same function as the `Ctrl-c` key sent from the terminal. It will usually terminate a program.
9	`KILL`	Kill. This signal is special. Whereas programs may choose to handle signals sent to them in different ways, including ignoring them all together, the `KILL` signal is never actually sent to the target program. Rather, the kernel immediately terminates the process. When a process is terminated in this manner, it is given no opportunity to "clean up" after itself or save its work. For this reason, the `KILL` signal should only be used as a last resort when other termination signals fail.
15	`TERM`	Terminate. This is the default signal sent by the `kill` command. If a program is still "alive" enough to receive signals, it will terminate.
18	`CONT`	Continue. This will restore a process after a `STOP` signal.
19	`STOP`	Stop. This signal causes a process to pause without terminating. Like the `KILL` signal, it is not sent to the target process, and thus it cannot be ignored.

Number	Name	Meaning
3	QUIT	Quit.
11	SEGV	Segmentation Violation. This signal is sent if a program makes illegal use of memory, that is, it tried to write somewhere it was not allowed to.
20	TSTP	Terminal Stop. This is the signal sent by the terminal when the `Ctrl-z` key is pressed. Unlike the STOP signal, the TSTP signal is received by the program but the program may choose to ignore it.
28	WINCH	Window Change. This is a signal sent by the system when a window changes size. Some programs , like `top` and `less` will respond to this signal by redrawing themselves to fit the new window dimensions.

`kill` Doesn't Always Mean Kill!

The command to send a signal is `kill`.

```
kill [-signal] PID
```

PID refers to the process ID of the recipient. This command is probably called "kill" because the default signal is TERM (Terminate). But a program can still ignore TERM (e.g. if broken). So if all else fails, try `kill -9`.

Note that only the owner of the process (or the superuser) can kill a process.

You can also kill all process belonging to a particular user or having a particular name with `killall`.

```
killall [-u user] [-signal] name...
```