Session 2 : Processes

This week we will learn about processes, their attributes, and how they are managed by the OS. The attributes of a process (e.g. process id, parent process id, priority, state, e.t.c.) are stored in the **process control block** (PCB). In this session, we will explore various Unix commands that give us insight into these attributes, as well as commands that give some control over various processes executing on the system. On completion of these activities you should be able to:

- Use the system tools **ps**, **pstree** and **top** to learn about processes.
- Use | to build pipelines.
- Use **grep** to filter text.
- Use **&** to run commands asynchronously.
- Use the **kill** command to send signals to processes.
- Write simple shell scripts.

Preliminaries

Shell variables.

Open the Fedora33 JSLinux shell or ssh into City's Linux server (see the instructions for lab 1). Shells such as Bash provide a mechanism for updating and using (dereferencing) variables. Variable assignment in the Bash shell has this form:

VARIABLE_NAME=*value*

where *value* is a string. Note that there are **no spaces** around the = sign; if you leave spaces, Bash will treat them as delimiters and try (but fail) to execute part of your statement as a command. To obtain the current value of a variable, you prefix its name with a $. Try the following in the shell:

```
SOMETHING=nothing
MESSAGE="$SOMETHING will come of $SOMETHING"
echo $MESSAGE
```

As you can see, strings are concatenated by simply putting them next to each other. The quotation marks in the second line are necessary because, otherwise, Bash tries to execute the word "will" as a command (try it without the quotation marks to see what happens). The quotation marks make the rhs of the "=" a string. The `echo` command prints any string it is given to the shell.

Bash also maintains some special read-only variables (the bash man page lists them in a section called "Special Parameters") some of which will be useful below. In particular, use `echo` to print:

$$ The process id of the current shell

$! The process id of the most recently executed background command

(Note: the variable *names* are `$` and `!`, respectively, while `$$` and `$!` are expressions which *dereference* those variables to give you their values as strings.)

Some shell tips.

- Command history: use the up and down arrow keys to move back through your command history to save typing the same thing over and over again.
- Filename completion: part way though typing a filename you can hit the Tab key for auto-completion; very useful for long file names.
- Aliases: you can create a short alias for a long command. For example, (see Task 1) you may find yourself typing `ps -e -o user,state,pid,ppid,command` many times. This gets annoying. You can make `pps` an alias for this long command as follows:

  `alias pps="ps -e -o user,state,pid,ppid,command"`

  Now you can just type **`pps`**.

Task 1: printing process attributes using `ps`

The primary Unix command for gathering information about processes is **`ps`**; an ancient command which has taken a variety of different forms on different versions of Unix. Modern versions of `ps` try hard to be all things to all people and, as a result, offer a bewildering number of options allowing the same information to be reported in many different ways. We will stick to just a few of the Unix single-dash-style options. Read the `man` page to see what you are missing.

Enter this: **`ps -e -o user,state,pid,ppid,command`** . The options here mean the following:

`-e` : show all processes (the default is to show only the processes which are owned by you and attached to your current shell);

`-o user,state,pid,ppid,command`: tailor the information displayed to include just the information you want; in this particular case: user name, process state, process id, parent process id, and (where possible) the command line string which launched the process.

You can trace a path in the tree of processes, working backwards from a "leaf" process (one with no children) to its first ancestor (a process with no ancestors) by reading the output from `ps` .

***Activity*:** find the user name, process id and parent process id of the `ps` process itself and write this information down; now find the line of ps output which corresponds to the *parent* process of the ps process and write down all the info for that process. Repeat until you get back to the mother of all user processes.

Activity: Run the `ps` command. From the processes listed, choose a leaf process. What is its first ancestor? How many processes are there in the complete path from the first ancestor to the leaf process?

Task 2: exploring process trees using `pstree`

In the previous task you used information from the `ps` command to determine a single path in the process tree leading from a leaf process (e.g. `ps` itself) back to the root process – the root process is called *init* and has process id 1. You may notice that `ps` displays the name of Process 1 as **`systemd`**, rather than *init*. This is because CentOS (the Linux distribution that City University uses) uses the systemd framework to initialise its OS services. In this framework, *init* is actually a symbolic link to the file **`/usr/lib/systemd/systemd`**.

There is a very useful system tool called `pstree` which will draw an entire process tree for you. Try this:

```
pstree -h -p
```

Read the `pstree man` page to see what the command line options mean.

Task 3a: monitoring process activity using `htop`

`pstree` is useful for gaining a snapshot of the executing processes and where they came from, but it doesn't tell us much about what they are doing. The command **`top`** (and its slightly more user-friendly variant, **`htop`**) plays a similar role for Linux as the Task Manager does for Windows, showing processes as they come and go and the demands that they make on system resources (such as CPU-time and memory) as they change over time.

Run `htop`. By default, `htop` lists the current processes in decreasing order of their % use of CPU time. If no particularly busy processes are currently running they will all be shown as using zero, or close to zero, CPU %.

Close `htop` by pressing q. Locate the shell-script files for today's tutorial on Moodle and upload them to your Linux shell (see last week's tutorial on how to upload). Make the file *IN1011Lab2_silent backgroundWork.sh* executable by changing its permissions:

```
chmod 755 IN1011Lab2_silent backgroundWork.sh
```

Execute this file as a "background" process:

```
./IN1011Lab2_silent backgroundWork.sh &
```

The *IN1011Lab2_silent backgroundWork.sh* file contains the code:

```
while true ; do : ; done
```

This is a while- loop which loops forever, doing nothing each time round the loop (the colon is a built-in "do nothing" command in bash, while the semicolons delimit commands). Now, start `htop` and look at the top of the `htop` window.

Question: What process is shown at the top of the list? Why is this process getting close to 100% of the CPU time?

Question: Close `htop`, create another background process that executes the shell script, then open up `htop` again. How do you explain what you now see at the top of the `htop` window?

To stop the shell scripts executing, issue the commands: **`kill %1 ; kill %2`**

The `kill` command can be used to suspend or terminate processes. Here, each process executing the shell script is a "job", identified by Unix via an id called the "jobspec". Jobspecs are assigned to processes running in the background of a given shell in the order these processes were spawned by the shell. So above, by executing the kill command, we terminate the 1st and 2nd jobs spawned by the shell. More on the `kill` command later in this tutorial.

[Optional] Task 3b: changing process priority using `renice`

The command `renice` allows you to change the scheduling priority for a running process. To some extent, you can use it to alter the behaviour of the **short-term scheduler** we learnt about in the lecture 2 videos. Recall that the short-term scheduler chooses which processes in the "ready" state will be assigned to the CPU next. A typical use of `renice` would be:

`renice -n` *priority pid*

where *priority* is any number between -20 (maximum priority) and 19 (minimum priority), and *pid* is the process id of the process to be modified. The default priority for processes that haven't been "reniced" is 0. See the man page for more information.

Note that `top` and `htop` display two columns, PRI and NI, relating to process priority; to a first approximation, NI is the priority as just described and PRI is simply NI + 20.

Activity: Repeat task 3a and use `renice` to modify the share of CPU time which the two shell-script processes receive. To do this, you will need to determine the process ids (not jobspecs) for these processes. By increasing the *priority* number for one of the processes (so, reducing its priority), experiment to find pairs of values which result in one of the processes receiving about 80% and the other about 20%.

Task 4: combining commands using "|"

A common way of combining commands in Unix is to use *pipelines*. A pipeline runs two commands concurrently and connects them by "piping" the output from the first command into the input of the second. Pipelines are formed by writing | between commands. (Note: there are two vertical bar symbols, so make sure you use "|"and not the broken pipe "¦").

For this task you are going to use the scripts `IN1011Lab2_mywords` and `IN1011Lab2_eatmywords` from Moodle. Using the `nano` command, have a look at the content of these files to get a rough idea of what they do (the first line in each script is known as a "shebang"; Google is your friend). Ensure the scripts executable (using `chmod` if necessary) and run the script `IN1011Lab2_mywords` by entering:

```
./IN1011Lab2_mywords
```

By the way, the leading `./` is necessary to persuade bash to look in the current directory for the program, which it won't do by default. Use the `ctrl+c` key combination to terminate the script when you get bored.

Now run `IN1011Lab2_eatmywords`. Nothing seems to happen. The script is waiting for input on its standard input which, for now, is whatever you type at the keyboard. Enter a few lines of text

to see how the script behaves. You can either terminate by using `ctrl+c` or by `ctrl+d` (which closes the script's input stream).

Now run the two scripts in a pipeline (in the background) by typing:

```
((./IN1011Lab2_mywords | ./IN1011Lab2_eatmywords) >> TheseWereMyWords.txt) &
```

The & means the long command above will execute in a separate process from the Bash shell. This long command passes the strings (i.e. the "word"s) produced by IN1011Lab2_mywords as input to IN1011Lab2_eatmywords. And the strings produced by IN1011Lab2_eatmywords are saved to the *TheseAreMyWords.txt* text file.

***Activity*:** Use the `ps` command to verify that both scripts are running as separate processes.

Question: The two processes have the same parent: which process is this? What is the relationship between this parent process and the Bash shell that you typed the command above in?

***Activity*:** Use the `kill` command to kill the job that these shell-script processes are part of.

Task 5: filtering strings using `grep`

The standard Unix system tools include a variety of powerful text processing tools. One of the most widely used tools is **`grep`**, which filters text. (Like many such tools, grep supports *regular expressions* for pattern matching, but in this exercise you will be using grep in only a very basic form). Suppose you know the name of a running process and you want to find its process id (you might want to kill the process, for example; see the next task). You can run

```
ps –e –o pid,command
```

and then scroll through the output to find any lines matching the name you are interested in. But as you have seen, `ps` will often report a very large number of processes, and it may be hard to find the one you want. A simple improvement is to use a pipe and grep to filter out just the lines of interest. Try this:

```
ps –e –o pid,command | grep sbin
```

The default behaviour of grep is to read lines of text from its standard input stream, echo any lines which contain a match for the specified pattern ("sbin" in our example), and discard the rest. Since we have piped the output from the ps process into the grep process, we only see the lines which grep selects and echoes.

Now do the same again to find the process id of the **`netns`** process (a process for creating and managing network namespaces. You might see usages of this in IN2011).

Task 6: running processes in the background

Run the `IN1011Lab2_sneeze.sh` script from Moodle:

```
./IN1011Lab2_sneeze
```

This outputs "sneezes" to the shell. Notice that you cannot enter any further commands in the parent shell while this command is executing[1]; the default behaviour of bash is to run a child process *synchronously*, i.e., to wait until it finishes. You can terminate the shell script with `ctrl+c`.

We would often like to execute commands *asynchronously*, which just means that the parent process is allowed to continue concurrently without waiting for a child process to finish. This can be done by putting an ampersand (&) after the command name, like so:

```
./IN1011Lab2_sneeze &
```

Do this now. This is referred to as running a command "in the background".

Unfortunately, the output from "sneeze" is very distracting, making it difficult to use the parent shell even though "sneeze" is running asynchronously. You want to stop the sneezing but `ctrl+c` doesn't work with background processes (try it). The `fg` command (short for foreground) can be used to make a background process the process that you directly interact with; i.e., the foreground process.

Run the command

```
fg
```

You are now in the shell that `IN1011Lab2_sneeze.sh` is running in, and so you can terminate `IN1011Lab2_sneeze.sh` with `ctrl+c`. Using process ids, `fg` can be used to bring a specified background process into the foreground.

There is another way to terminate background processes using a form of inter-process communication. We can send a termination *signal* from a foreground process, using the id of the process we wish to terminate. Invoking the `kill` command is how we do this:

```
kill –SIGTERM pid
```

Task 7: ordering process executions using `kill`

Despite its name, we can use kill to send signals which *don't* kill the receiving process. For example, the signals SIGSTOP and SIGCONT, respectively suspend a process (putting it into a wait state) and resume it (putting it into the ready state). We referred to the "wait" state as a "suspend" state in the lectures.

***Activity*:** Execute *IN1011Lab2_silent backgroundWork.sh* in the background. Use SIGSTOP and SIGCONT to control this process, rather than terminating it with SIGTERM.

Question: Using `ps` and `grep` to find out, what effect do these the signals have on the process state reported by `ps` for *IN1011Lab2_silent backgroundWork* (for reference with the lectures, R: running or ready, S: blocked, T: suspended).

Task 8: shell scripting practice

Now run `IN1011Lab2_sneeze.sh` and `IN1011Lab2_blessyou.sh` concurrently (but not in a

[1] Well, you can, but bash will ignore what you type until the new process exits.

pipeline) by entering:

```
./IN1011Lab2_sneeze.sh & ./IN1011Lab2_blessyou.sh &
```

Note how the output alternates so that it seems like "blessyou" is responding to each sneeze. Actually these scripts are cheating; the alternation is achieved simply by delaying the start of the "blessyou" sequence by one extra second. Locate the use of `sleep` in the "blessyou" script which does this. While this sort-of works, it is not robust, because the two scripts are actually operating completely independently, rather than in synchrony.

***Activity*:** Improve the scripts as follows:

1. Add a line to the "sneeze" script (before the while loop) so that "sneeze" runs "blessyou" in the background. Now you can launch the pair just by entering:

   ```
   ./IN1011Lab2_sneeze.sh &
   ```

   Check that this works as expected.

2. Modify both scripts to use kill to coordinate their behaviour. When you get this right, you should be able to temporarily stop and resume "sneeze" and "blessyou" without the message sequences getting out of sync.

   Hint: "sneeze" can use the special shell variable **`$!`** to find "blessyou"s pid, while both "blessyou" and "sneeze" can use **`$$`** to find their own pids, and "blessyou" can use **`$PPID`** to find "sneeze"s pid.