ECS518U Operating Systems

Lab 2: Process monitoring in Linux

This exercise is not assessed (but you must get it ticked off by the demonstrators)

Preliminary

Aim

The aim of this lab is to find out how you can discover information about processes in Linux systems (state, memory requirements and usage, running time, etc.). The main commands we will use in this lab are:

- `ps` (process status), `pgrep`, `pkill`, `kill`, `killall`
- `top` (an interactive, real-time view of processes running on a system, together with all sorts of useful information about the processes).
- `pstree` (process information in tree form showing parent/child relationships)

You should be able to:

1. Open a terminal in Linux and use commands (e.g. those from Lab 1) to navigate in your directories and examine your files and their properties (e.g. r-w-x permissions).
2. Read the man pages of commands when you need to find out information about specific options, e.g. `man ps` and find out about commands in online tutorials and other help sources.

This lab sheet is not meant to act as a stand-alone tutorial. It gives you some basic information about the commands but it is up to you to explore further how you can get the commands to work for you (e.g. using man pages, searching online, or simply playing with the commands in a Linux terminal).

Note also that output formats may vary slightly between machines and shells, and will most likely vary if you try these commands on a Mac or certain distributions of **Ubuntu.**

1 The `ps` (process status) command

The `ps` command (process status) gives various types of information about processes running. Depending on what options you supply to the command you can select different processes to view and you can display different types of information for these processes. It supports searching for processes by user, group, process id or executable name.

a. List all processes in current shell (terminal window)

If you simply run `ps` without any other arguments, you will get a list of processes for the current running shell (i.e. from the terminal you ran the command from), something like:

```
[tassos@bert ~]$ ps
  PID TTY          TIME CMD
19778 pts/4    00:00:00 ps
28183 pts/4    00:00:00 tcsh
```

where PID is the unique process id, TTY the console (terminal) that the user is logged into, TIME the amount of time the process has been running and CMD the name of the command that launched the process.

b. Display all processes running on a system

To list all processes for a system, use the –e option (typically the output will be very long so may want to pipe to a paginator such as `more`):

```
[tassos@bert ~]$ ps -e | more
   PID TTY          TIME CMD
    1     ?        00:01:39 init
    2     ?        00:00:01 kthreadd
    3     ?        00:00:03 migration/0
    4     ?        00:00:03 ksoftirqd/0
    5     ?        00:00:00 stopper/0
….
```

Note that you get the same output with issuing the command: `ps -A.`

A useful hit: You can **redirect** the output of any command in Linux and instead of appearing on the screen, it can be saved on to a file, for example:

`ps -e > ps_output.1` will save the output of the command in the text file ps_output.1. You can open and examine the file. Check **http://linuxcommand.org/lc3_lts0070.php** for redirection in Linux, it is very useful to know.

If the `TTY` value is set to "`?`" this means that the process is not associated with any user terminal (e.g. can be a system process).

More useful information can be given if you combine this with the –f (or –F) option:

```
[tassos@bert ~]$ ps -ef | more
UID          PID    PPID  C    STIME      TTY    TIME        CMD
…
root         16     2     0    Jan07       ?    00:00:00 [stopper/3]
root         17     2     0    Jan07       ?    00:00:03 [ksoftirqd/3]
root         18     2     0    Jan07       ?    00:00:02 [watchdog/3]
…
tassos       879    28183 0    14:24       pts/4  00:00:00 ps -ef
tassos       880    28183 0    14:24       pts/4  00:00:00 more
tassos       28177  27800 0    13:59       ?     00:00:00 sshd: tassos@pts/4
tassos       28183  28177 0    13:59       pts/4  00:00:00 -tcsh
….
```

Here we see the addition of another useful piece of information for a process: in the `PPID` column we see the **id of the parent process**. For example, the parent process id of the command I ran in this example "`ps -ef | more`" is 28183 – and it corresponds to the terminal window (shell) that I ran the command from (make sure you can see this in the example above) - `tcsh`.

You can combine any of these commands with the `grep` command using pipes – `grep` looks for a pattern in a text file or a text stream and returns it as its output. For example:

`ps -ef | grep tcsh` in the example above will return the line of the output that contains the strong 'tcsh:

```
tassos      28183  28177 0   13:59       pts/4  00:00:00 -tcsh
```

This is a very useful combination and one you should get used to. `grep` is a very powerful utility in Linux.

An alternative way to use the ps command using the BSD syntax (Berkely Standard Distribution) is without "-" before the various options. A very popular format is: `ps aux:`

```
[tassos@bert ~]$ ps aux
USER       PID %CPU  %MEM    VSZ   RSS  TTY      STAT START   TIME COMMAND
tassos    2500  0.0  0.0 122680  1244 pts/3    R+   11:44   0:00 ps aux
tassos   18114  0.0  0.0 128848  2036 ?        S    11:41   0:00 sshd:
tassos@pts/3
tassos   18121  0.0  0.0 125076  2040 pts/3    Ss   11:41   0:00 -tcsh
```

The **`STAT`** column gives information about the **state** in which each process in the list currently is (link to process states and transitions from the lectures):

- D Uninterruptible sleep (usually IO)
- R Running or runnable (on run queue)
- S Interruptible sleep (waiting for an event to complete)
- T Stopped, either by a job control signal or because it is being traced.
- W paging (not valid since the 2.6.xx kernel)
- X dead (should never be seen)
- Z Defunct ("zombie") process, terminated but not reaped by its parent.

And the additional characters you may see in the column:

< high-priority (not nice to other users)

N low-priority (nice to other users)

L has pages locked into memory (for real-time and custom IO)

s is a session leader

l is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)

+ is in the foreground process group.

VSZ (Virtual Set Size) and RSS (Resident Set Size) are linked with memory management and will make more sense after the lectures in weeks 6 & 8.

You can find all the above information + much more in the man pages for `ps` and very easily anywhere online by using a search engine and searching e.g. for "what is the VSZ column in ps command".

Other common and useful combinations of options are: `ps –def` and `ps –elf` (explore them and try to make sense of the information in the columns).

c. Display processes running for a specific user

We can combine the previous command with the –u <user_id> option and display information for processes running by a specific user, as in the example below:

```
tassos@bert ~]$ ps –fu tassos
   UID        PID          PPID  C STIME   TTY      TIME         CMD
tassos       17979        28183 0 14:34   pts/4    00:00:00 ps -fu tassos
tassos       28177        27800 0 13:59     ?      00:00:00 sshd: tassos@pts/4
tassos       28183        28177 0 13:59    pts/4   00:00:00 -tcsh
```

You can get a longer listing with details about processes with the `–l` option:

```
[tassos@bert ~]$ ps –lfu tassos
F S UID      PID   PPID  C  PRI  NI  ADDR  SZ      WCHAN   STIME     TTY     TIME      CMD
0 R tassos  18831  28183 0  80   0   -     30667     -       14:56     pts/4   00:00:00 ps -lfu tassos
5 S tassos  28177  27800 0  80   0   -     32201     poll_s  13:59     ?       00:00:00 sshd:
tassos@pts/4
0 S tassos  28183  28177 0  80   0   -     30746     rt_sig      13:59     pts/4   00:00:00 -tcsh
```

Some interesting fields are: **PRI** (the priority of the process for scheduling), **NI** (the 'niceness' of the process, also linked with priority - the higher the value, the lower the priority. The default nice value is 0 on Linux systems.). **SZ** displays the size of the process in memory. The value of the field is the number of pages the process is occupying. On Linux systems a page is typically 4,096 bytes.

Useful links to material we will cover in Weeks 2 & 3 for processes & scheduling

WCHAN (wait channel) is the name of the kernel function in which the process is sleeping (waiting) (or – if the process is running, like for process with PID 18831 above). When a process puts itself to sleep, it is voluntarily giving up its turn on the CPU (voluntary yield).

For example, rather than constantly checking the keyboard for input, the shell puts itself to sleep while waiting on an event. That event might be an interrupt from the keyboard. When a process puts itself to sleep, it sleeps on a particular wait channel (WCHAN). When the event that is associated with that wait channel occurs, every process waiting on that wait channel is woken up. There is probably only one process waiting on input from your keyboard at any given time. However, many processes could be waiting for data from the hard disk. If so, there might be dozens of processes all waiting on the same wait channel and all are woken up when the hard disk is ready.

When a process puts itself to sleep, it is voluntarily giving up the CPU. It may be that this process had just started its turn when it noticed that it didn't have some resource it needed. Rather than forcing other processes to wait until it gets its "fair share" of the CPU, that process is nice and lets some other process have a turn on the CPU. The scheduler reciprocates the niceness by allowing that process to set its own priority at which it will run when it wakes up.

There are numerous other options, some allowing to filter processes by user group, pid, etc., some others allowing to display the process hierarchy in a tree style, some giving more specific information about memory usage, etc.

2 pgrep

The `pgrep` command is based on a very popular utility, `grep`, which searches for a pattern in e.g. a text file, and then outputs matches to that pattern. `pgrep` allows users to look up process information based on e.g. name, owner and other information.

For example, to find **the pid of a specific process** (e.g. `tcsh` - of the shell I am using in as many terminal windows as I have open) owned by me: `pgrep tcsh –u tassos`

If you want **to get more information than just a pid for the matching list of processes**, you can add the `–l` option: `pgrep tcsh –u tassos –l` will output something like:

```
18114 sshd
18121 tcsh
```

If you want to use the command to print out **the number of processes owned by a specific user**, you can do it by: `pgrep –u tassos -c`

To get the complete path to the command that corresponds to the matching process, you can add the –a option: `pgrep –u tassos –a` will output a long list of processes, like:

```
2160 /usr/bin/firefox
2167 /usr/lib/gvfs/gvfsd
```

etc.

More options can be found in the man pages and online.

3 Killing processes: kill and pkill

In Linux you can terminate a process by sending a specific **signal** to it. The 'act' is called killing a process and commands that make this happen include `kill` and `pkill`.

(**Signals** are a powerful way through which processes can communicate in Linux and we will see a little bit more about signals in later labs. Anyone interested can look at e.g. https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html).

There are of course certain **rules** when it comes down to killing processes: A user can kill any of the processes owned by him/her, but not those owned by other users. A user can also not kill any system processes. The **root** user can kill pretty much anything ☺.

In order to kill a process with the `kill` command, we must know its pid (we saw ways of getting this information in previous sections). For example, `kill 1234` will kill the process with pid 1234. In terms of signal passing, this is sending the **SIGTERM** signal to the process and is asking it to terminate. When we press ctrl and c on the keyboard to terminate a process we are essentially sending it the **SIGTERM** signal

If a process does not want to be killed ☺ we can force it, by sending the **SIGKILL** signal – signal no 9 for anyone who looks this up – and we could do this by: `kill -9 1234` . This is a rather extreme solution and may lead to unstable exit (e.g. loss of data).

We can use the **`pkill`** command to kill processes by name, e.g. `pkill firefox`

We can use the **`killall`** command to kill all instances of a process, by name. For example, `killall firefox` will kill all instances of the `firefox` process running at that time and owned by me.

4. Other commands worth knowing about

`pstree`

(**NOTE: `pstree`** is not installed by default in MacOS. You can always try it on a Linux desktop in the ITL / Electronics Lab, or else you can install the utility using **homebrew** (`brew install pstree` should do it).

The `pstree` command is similar to `ps` in that it can be used to show all of the processes on the system along with their PIDs. However, it differs in that it presents its output in a *tree diagram* that shows how processes are related to each other and in that it provides less detailed information about each process than `ps`. As you can imagine the whole hierarchy tree of all the processes running on a system can be very long so you can search for specific users of processes / programs with command line options (for example `pstree -U tassos` would display the part of the tree that has processes owned by user `tassos`).

```
tassos@bert ~]$ pstree
init-+-VGAuthService
     |-acpid
     |-atd
     |-auditd-+-audispd-+-sedispatch
     |        |         `-{audispd}
     |        `-{auditd}
     |-automount---10*[{automount}]
     |-avahi-daemon---avahi-daemon
     |-11*[bonobo-activati---{bonobo-activat}]
     |-chronyd
     |-7*[clock-applet]
     |-console-kit-dae---63*[{console-kit-da}]
```

```
      |-crond
      |-cupsd
      |-15*[dbus-daemon---{dbus-daemon}]
….
```

top

`top` is an interactive tool that allows monitoring the processes running on a system. It displays one line per process with various columns that contain CPU related, memory related, or more general information and it is more similar to interactive GUI-based tools for process monitoring (you can for example choose which field to sort processes by). A very good guide is available at: https://www.booleanworld.com/guide-linux-top-command/, or https://www.geeksforgeeks.org/top-command-in-linux-with-examples/.

```
top - 15:49:15 up 14 days,  2:51, 16 users,  load average: 0.16, 0.47, 0.52
Tasks: 662 total,   1 running, 659 sleeping,   2 stopped,   0 zombie
Cpu(s):  9.4%us,  3.5%sy,  0.0%ni, 87.2%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8060472k total,  7386344k used,   674128k free,   406180k buffers
Swap: 10485756k total,    17800k used, 10467956k free,  3035640k cached

PID    USER          PR      NI      VIRT  RES   SHR S       %CPU    %MEM    TIME+  COMMAND
5247   xyz1          20      0       888m  23m   15m  S      2.0     0.3     29:07.09 knotify4
11358  xyz2          20      0       832m  21m   14m  S      2.0     0.3     29:44.47 knotify4
12531  xyz3          20      0       888m  23m   15m  S      2.0     0.3     29:48.58 knotify4
3320   root          20      0       145m  9.9m 1336  S      1.6     0.1     51:02.82
x2gocleansessio
12762  tassos        20      0       27892 1928 1124  R      1.6     0.0     0:00.56 top
24284  xyz4          20      0       1262m 273m  46m S      1.3     3.5     30:58.76 firefox
2187   root          20      0       1038m 9020 3244  S      1.0     0.1     2:04.42 automount
2030   root          20      0       117m   24m 2508   S      0.7     0.3     61:35.14 mcollectived
```

Note the mention of **zombie processes** in the output of top above (Week 3).
The amount of information given will vary across systems (information used here has been taken from: https://www.unixtutorial.org/commands/top/). Interesting items to note:
Cpu(s):

- *us – User CPU time*. The % of time the CPU has spent running users' processes with default priorities
- *sy – System CPU time*. The % of time the CPU has spent running the kernel and its processes
- *ni – Nice CPU time*. The % time the CPU has spent running users' process that have been prioritized up using nice command
- *id* – Idle.
- *wa – I/O wait*. The % of time the CPU has been waiting for I/O operations to complete
- *hi – Hardware IRQ*. %of time the CPU has been servicing hardware interrupts
- *si – Software Interrupts*. The % of time the CPU has been servicing software interrupts
- *st – Steal Time*. The % of CPU 'stolen' from this virtual machine by the hypervisor for other tasks (such as running another virtual machine) – a fairly recent addition to the top command, introduced with the increased virtualization focus in modern OSs.

From the fields of information displayed for each process we have seen some of them previously. Some others worth noting (but more related to memory management so just keep them in mind for later when we revisit the topic during weeks 6 & 8):

- **VIRT** – the amount of virtual memory used by a process: code, data and shared libraries plus pages that have been swapped out
- **RES** – the resident part of a process – how much of it resides in the physical memory.
- **SHR** – shows you the size of potentially shared memory segments for a process
- **S**: Process state (list of possible process states has been given in a previous section).

Tasks for you to do & Answer Sheet

MacOS users: please read the information in the Lab2 section on QMPlus about some of the tasks described here and see notes in some of the tasks.

1. **Experiment with the commands given in the sheet**
   - Run the examples from a Linux terminal, try variations and experiment with different options. Keep in mind that if you use a Mac or certain versions of Ubuntu, output may vary from the examples given.
   - If some of the output or command options are not clear:
     - Look for help through the `man` pages, or online
     - Ask for help during your lab time
2. **Simple ps command**

   Open two terminal windows. In one of them start the firefox browser by typing `firefox &` - include the & character: this will run firefox and make the terminal available for further use – runs the process in the background. Run the command `ps` from the same terminal and from the second terminal.

   NOTE: If you have firefox already open in your system e.g. via the menus, this task will not work. Close all other instances of firefox and make sure the first one opening is the one in this task.

   For Mas Users: use `/Applications/Safari.app/Contents/MacOS/Safari &` instead of `firefox &`

   In which terminal window can you see the firefox process in the ps output? Why?

   In both outputs you will see the shell process (`bash` or `tcsh` (or `zcsh` in Mac)). It does not have the same pid, even if it has the same process name. Why?

   In the window you ran the firefox command from, type: `ps –Oppid` (O is the capital letter, not the number) – this gives you a neat view of process id and parent pid for processes in the current terminal window. You will most likely see a few processes with the firefox name.

   Identify which is the "main" firefox process (hint: try to look at the ppid information to do this).

   Which process is the parent process of the main firefox process?

3. **Working with two terminal windows.**

   Open two terminal windows. From one terminal run "`gedit`" (it is a text editing application). You will see that this terminal window is now "tied up" with running `gedit`. From the other terminal, try to find out the process id of `gedit`.

   gedit pid:

   Mac users: use `nano` instead of `gedit`

   Command(s) you used to get this information:

   Now kill `gedit` from the second terminal (i.e. not from the one you ran the command from). Show two different commands that can do this:

4. **How many of the running processes are "owned" by user "root"?** (do not count yourselves, have a command do the counting for you!!!)

   How many running processes are owned by you? (do not count yourselves, have a command do the counting for you!!!)

   What command(s) did you use to find out these answers?

   Can you think of a way to answer the same question by using pipes and some of the commands we saw in Lab 1 (e.g. `wc`)?

5. **What is the 'root' user in Linux systems?**

   What is the '`init`' process in Linux systems (nowadays called '`systemd`', or if you are using MacOS then it is `launchd`)?

   What command can you use to find out the process state for the process `init` (`system` or `launchd` in MacOs)? What is the process state?