Unix Fundamentals – Processes and Signals

Marek Kozłowski
Faculty of Mathematics and Information Sciences
Warsaw University of Technology

Never use copy&paste for the following exercises. Retype all commands manually!
Don't just read those exercises and examples. Do them, please!

1. **Processes**

   (a) Read about the '`fork()`' system call:
   `$ man 3p fork`
   Remember: Reading POSIX man pages (3P) is a good practice for system calls!

   (b) How many processes are created by calling:
   `for (i=0; i<4; i++) fork();`
   No, '4' is certainly the wrong answer!

   (c) Let's check your answer for the previous question. Open *geany* and enter the following code:

   ```
   #include<unistd.h>
   #include<stdio.h>
   int main()
   {
     for (int i=0; i<4; i++) fork();
     printf ("!");
     return 0;
   }
   ```

   Save this code as a C file. Built it by clicking the *brick* and run by clicking the *gears* (or from a terminal). Was you answer correct? If not – try to analyze the code again.

   (d) Note that in fact there is no such system call as '`exec()`':
   `$ man 3p exec`
   However all those do the same work. The difference is in formal parameters.

   (e) Compile the following code:

   ```
   #include<unistd.h>
   int main()
   {
     execl("/usr/bin/ls", "ls", "/etc", (char*)0);
     return 1;
   }
   ```

   What is the exit status? Let's check:
   `$ ./program_name && echo success`
   Yes, it is success (0). Why? Because the '`exec()`' system call overwrites the whole process image.

   (f) Display the whole process tree:
   `$ pstree -p`
   Remember that names in curly braces are threads, no processes!

   (g) List all processes of the current session:
   `$ ps -f`
   What is the parent process ID for this bash instance?

(h) List all processes:
`$ ps -ef`
Names in square brackets are mostly kernel threads.

(i) The same using BSD syntax:
`$ ps aux`
Don't place a dash before options!

Students' login shell are set to ***`mini`****. On lab workstations it is a symlink pointing to the* ***`bash`*** *executable. The process stores only one name – the one specified when executing it – which is in our case* ***`mini`****.*

(j) What processes execute '`bash`'?
`$ pgrep mini`

(k) The same as above with '`ps`' and '`grep`':
`$ ps -ef | grep mini`
Oups! A small correction is required:
`$ ps -ef | grep mini | grep -v grep`

(l) Start '`top`'. By pressing '`<`' and '`>`' sort processes by CPU utilization.

(m) Take a look at *procfs* file system definition. Read:
`$ man 5 proc`
Check files representing one of your processes.

2. **Signals**

(a) Start with:
`$ man 7 signal`
What are default actions?
What signals are defined by POSIX.1-1990?
What is the most common action?
What is the only signal with the default action *ignore*?
What signal is sent on incorrect memory reference? What happens if you divide by 0?
What number is assigned to '`SIGKILL`'? Memorize that number.
What signals cannot be blocked, ignored nor caught?

(b) Start '`top`'. Terminate it by pressing '`Ctrl-C`'. Yes, you've just sent the '`SIGINT`' signal to it.

(c) Start '`top`' again. Open another terminal emulator and run:
`$ kill -SIGINT `pgrep '^top$'``
Yes, '`SIGINT`' may be sent that way too.

(d) Start '`top`' again. In another terminal emulator run:
`$ kill -SIGFPE `pgrep '^top$'``
Well, honestly, this signal is not intended for such use but you may generate any signal synthetically with '`kill`'.

(e) Start '`vim`' in your current terminal emulator window. Open another terminal and execute:
`$ kill `pgrep vim``
Repeat the same experiment but this time use '`SIGKILL`' instead of '`SIGTERM`':
`$ kill -SIGKILL `pgrep vim``
or just (you remember the number, don't you?):
`$ kill -9 `pgrep vim``
Unfortunately, due to some terminal emulator program bugs scrolling a mouse wheel over the window in which *vim* was used may generate some garbage. It's a bug; it has nothing to do with signals.

On systemd Linux workstations (we have in labs) the next point may lead to breaking a d-bus instance (whatever it is) which may result in making the system unstable. Ask the teacher before proceeding.

(f) If your GUI session freezes you may terminate it by switching to some tty (for `tty2` press '`Ctrl-Alt-F2`') and execute in it:
`$ killall -u `whoami``
or – if really necessary:
`$ killall -9 -u `whoami``
Note that terminating processes with '`SIGKILL`' is always risky. It may result in data loss or/and configuration improperly saved!

(g) Let '`cat`' reads from *stdin*:
`$ cat`
Type in some text and press '`[Enter]`'. Repeat this step several times. Then press '`Ctrl-D`'. It's not a signal. It just places an `EOF` character in the input stream.

(h) Start *thunderbird* from your terminal emulator:
`$ thunderbird`
and kill it by pressing '`Ctrl-C`'.
Now start *thunderbird* from your terminal emulator as a background process:
`$ thunderbird &`
The sequence '`Ctrl-C`' doesn't work for background processes. However we can use '`kill`':
`$ kill -SIGINT `pgrep thunderbird``

(i) Once again start *thunderbird* from your terminal emulator:
`$ thunderbird`
and stop it (more precisely: *freeze*) by pressing '`Ctrl-Z`'.
Now resume it as a background process in the current session:
`$ bg`
display its job number
`$ jobs`
move it to the foreground
`$ fg`
and kill it by pressing '`Ctrl-C`'.

3. **Process tracing**

(a) Start and trace '`gvim`':
`$ strace gvim`
Unfortunately it traces only a main process (which ends immediately after initialization is complete) not its children.
Trace the main process as well as its forks:
`$ strace -f gvim`
If you can't read fast enough save the output to a file:
`$ strace -f -o gvim.log gvim`
Can you see that there is a process group?
After you exit check how many system calls were used:
`$ wc -l gvim.log`
Check what files from '`/etc`' and in what order were opened by '`gvim`':
`$ grep open gvim.log | grep /etc`
and what icons were used:
`$ grep open gvim.log | grep png`
Finally trace each process to a separate file:
`$ strace -ff -o gvim.log gvim`
`$ ls gvim.log*`