System Software
Linux command line manual: lab 2
2016 - 2017

Bachelor Electronics/ICT

Course coördinator: *Luc Vandeurzen*
Lab coaches: *Floris de Feyter*
Stef Desmet
Tim stas
Jeroen Van Aken
Luc Vandeurzen

Last update: January 26, 2017

Lab targets: job control basics, Bash wildcards

Job control

If you run a program (also called a 'job') for a long time or you wish to start multiple programs concurrently without opening many terminals, you should run these programs in background.

- <program> & : the '&' indicates that the program should start and run in background;
- <ctrl>-z : pressing <ctrl>-z forces the currently running process to suspend mode; remark that <ctrl>-c is terminating a program – not suspending it; once the program is suspended, you can use the 'bg' or the 'fg' command;
  - bg : the most recent suspended program starts running in background;
  - fg : the most recent suspended program starts running in foreground; hence, with <ctrl>-z and 'fg' you can pause and resume programs;
- jobs : lists all jobs running in background in the terminal;
  - jobs -l : also provides the process ID (more on processes and process IDs follows later on in this course) of the job and info on how the jobs was stopped;
- fg %<job> : get the job with job <job> (the number in the ranking of the 'jobs' command) in foreground;
  - %<job> : is a shortcut for 'fg %<job>;
- kill %<job> : terminate the job <job> (the number in the ranking of the 'jobs' command);
  - kill <processID> : has the same effect; <processID> is obtained with (jobs -l';
- timeout <time> <program> : starts <program> and if the program is still running after <time> it will be stopped;
  - timeout 10s <program> : run the program for at most 10 seconds (m = minutes, h=hours, d=days);
- watch <program> : execute <program> periodically (default 2 seconds); some examples:
  - watch -n 10 <program> : run <program> every 10 seconds;
  - watch -d <program> : highlights the differences between two successive updates;

- wait <%list-of-jobs> : wait until all jobs in <%list-of-jobs> are finished;
    - wait <list-of-PIDs> : same but now for a list of process IDs;

As usual you can find more info on the usage of these commands in the man pages. However, man pages on the commands 'jobs', 'bg', and 'fg' might be missing. These are Bash commands, which means you will find more on them in the Bash documentation (e.g. man bash).

Exercise: *job control*

Go to one of your directories containing a 'main.c' file. Open this file with 'nano'. Change the first '#include' into 'include'. Now let's quickly find out what 'gcc' says about this. Don't close 'nano', but suspend it, then compile and observe the result, finally switch back to 'nano' to restore the original file.

Exercise: *job control*

This exercise requires that the C programming exercise on the temperature sensor node simulator is finished.

- Open two terminals. Start the sensor node simulator in background in one terminal and then use 'jobs' in both terminals and observe the results.
- The sensor node simulator doesn't stop running. You could use <ctrl>-c to stop it. Does it work?
    - Maybe try to get it running in foreground and then use <ctrl>-c?
    - Maybe try to use the 'kill' command?

Bash shell features: wildcards

Imagine you only want to list all files that end on '.c'. This could be done using 'ls' with wildcards, i.e. 'ls -l *.c'. Wildcards can be used in strings for pattern matching. Pattern matching is typically used in listings, file searching and manipulation (see another lab), data filtering, etc. As such, many commands can take advantage of using wildcards. To give another example, 'rm *.o' removes all object code (.o) files. A string pattern with wildcards will be expanded to a list of strings that match the pattern. Some examples:

- ? : match any single character, e.g. 'pts/3' satisfies the pattern 'pts/?';
- * : match any number of characters (zero or more chars), e.g. 'ma*.c' matches all c files starting with 'ma';
- [ ] : match any character specified with [ ], e.g. 'm[a,o,e]t' will expand to 'man', 'mot', and 'met';
    - - : is used to denote a range, e.g. 0-9 or a-z;
- [!] : logical not of [ ] : match any character that's not listed in [ ];
- { } : enumerates several patterns of which at least one must be satisfied, e.g. {*.odt, *.doc} matches all .odt or .doc files.

Notice that '\' is used as an 'escape' character to protect special characters like ?, , !, etc. For example, if you wish to indicate the file name 'ma*p' in a string pattern where the '*' should not be interpreted as the wildcard '*', then you should use 'ma*p'.

More details can be found in the man pages using 'man 7 glob'.

It's worth introducing here the 'echo' command. The 'echo' command seems not to be of much value since it just displays the text message following the command, e.g. 'echo hello world!' is sending the text 'hello world!' to 'stdout'. Reading the man page you will discover that you can add some formatting options, but basically that's it. Nevertheless, the 'echo' command is a widely used Linux command. The 'echo' command is typically used in shell scripts – which are out of the scope of this course – to display informal messages to the user while executing the script. However, the 'echo' command offers an interesting feature concerning wildcards. You can use 'echo' to print out what would have been executed when a command uses wildcards. For example, 'rm -R *' is a dangerous command; before executing it, you should use 'echo rm -R *' to get a better idea what exactly will match the '*' wildcard.

Exercise: *wildcards*

Go to '/etc'. Use 'file' to find out the file type of the following subset of files. Don't forget to use 'echo' to inspect the wildcard expansion.

- All files that end on ~
- All files that start with the char 'x' or 'y'
- All files that contain the substring '-n' with 'n' some 1 digit number

Exercise: *wildcards*

Go to your home directory. Compare the different results of the commands:

- ls *
- ls .*

Important remark

The discussion on advanced Bash shell features below is not considered as mandatory. First do your C programming exercises and only when you still have time left, start looking at these features.

Advanced Bash shell features

The following features are not something you will need all the time, but nevertheless it's interesting to know they exist.

- A bit more on I/O redirection. To better understand the following, you must know that 'stdin', 'stdout' and 'stderr' are associated by default to the file descriptors 0, 1 and 2 respectively.
    - 1> : redirect 'stdout' output to file; actually, this is he same as '>' because if you use output redirection without file descriptor, the default file descriptor '1' is assumed;
    - 2> : redirect 'stderr' output to file;

- &> : redirect 'stdout' and 'stderr' output to file; you may also use '>&' which is exactly the same;
- Note: the above also works on the append-redirection, i.e. '2>>' and '&>>';
- | tee : pipe output of the command to both file and stdout; for example:
  - ls | tee out : prints 'ls' output to screen (stdout) and to the file 'out';
- { cmd1; cmd2 ; } > some_file : (mind the spaces and the ';') redirect the output of the commands 'cmd1' and 'cmd2' to the file 'some_file';

- <cmd1> `<cmd2>` : command <cmd2> will be executed first and its output will be used as input for the command <cmd1>; remark that this looks a bit like using pipe-redirection '<cmd2> | <cmd1>' but it's not (always) the same;
- <cmd1> $(<cmd2>) : similar to the above;
- <cmd1> ; <cmd2> : execute command <cmd1> and only if this command is finished execute command <cmd2>; for example:
  - sleep 3; echo "test”
- <cmd1> && <cmd2> : execute command <cmd2> only if command <cmd1> has been executed successfully;
- <cmd1> || <cmd2> : execute command <cmd2> only if command <cmd1> has **not** been executed successfully;

Summary: list of commands

- bg
- echo
- fg
- jobs
- kill
- timeout
- wait
- watch