COMPUTER SCIENCE

Addendum to Assignment 1.

Processes.

GIEyAI
Fall Semester, 2022-2023
v2.7(07.09.22)

FURTHER TOPICS ON LINUX.

This document is an extension of Assignment 1, and goes a little further on a few basic topics in Linux. It is not a complete guide, but a simple presentation of some additional features in Linux which, while not necessary for Assignment 1, can be interesting for some students. It is prepared as a continuation of Assignment 1, so the student should complete it first.

ENVIRONMENT VARIABLES

In order to (among other things) simplify the use of the command interpreter, there are what are called the *environment variables*. These variables contain information about default work modes or places in the file system where things can be found. One of the most important ones is the PATH variable. It is just a list of directories in the file system where the command interpreter looks for the applications that he is told to run. Thus, if the PATH variables has directories `/bin` and `/home/larq/executables` (i.e.: `PATH=/bin:/home/larq/executables`), then if one types into a terminal:

```
bash-3.2$copy archivo1 archivo2
```

where `copy` is an application created by ourselves, then the command interpreter will look for it in all the directories present in the list of variable `PATH`, in order. First, it will look in `/bin`. If there is an executable file called `copy` there, it will run it. Otherwise, it will look next in `/home/larq/executables`, and so on. If it can't find any file called `copy` in any of the directories, it will say so, and stop. The search follows the order the directories are listed in in the variable, so if there are two versions of `copy` (obviously in different directories), it will run the first one it finds.

The PATH variable is very useful when one wants to arrange system (or personal) applications in an ordered way (i.e., each one in its specific place) in different places and use them without having to constantly change the current directory.

In general, one can define state variables for whatever purpose. Some are created by the system, like the PATH variable. Their value can be visualized with the echo command:

```
bash-3.2$echo $PATH
```

If the variable has the value seen above, this command will output:

```
/bin:/home/larq/ejecutables
```

It is also possible to modify them. For instance, if one is develping software (like in this lab, later on), it can be very convenient to have the command interpreter look first in the working directory (which it does NOT do by default). The following command achieves this:

```
bash-3.2$export PATH=.:$PATH
```

This command would set (`export`) an environment variable called `PATH` to the value resulting from concatenating `.` and the former value of the variable (`$PATH`). The dot (`.`) represents the working directory (whichever it is at the time the variable is used). The `export` command can be used to set variables of any kind. If the string which is going to be set as its value has blanks, then it must be between quotation marks.

```
bash-3.2$export VARIABLE="A string of characters"
```

PROCESSES

A multitasking operating system, such as Linux, can handle several processes executing simultaneously. To understand how this works, it is important to bear in mind what a process is. A process is an instance of a program in execution. The distinction between a program and a process is important because Linux allows execution of several instances of the same program simultaneously. A program is a software package which resides in the hard disk, for instance, and which, upon execution, is loaded into the system memory and run. This running code, together with its data and environment variables, is a process. One example: when one launches a terminal, the software package that is the terminal is brought from the hard drive to the system memory and is executed. A *'terminal'* process is in execution. Then, it is possible to give it some commands, which will be executed by the command interface, as we saw in the previous section. At this moment, one can launch another terminal and run some other commands in it. If so, the operating system will again load the software package that makes up the terminal program and run it, in a separate action from the one which ran the previous instance of the terminal. The second invocation of the terminal will be a different process. Even if the code is exactly the same, the data and the environment variables are not (or may not be). In this situation, with both terminals running, each terminal 'window' represents a different instance of the same program, the 'terminal program'. Each window (and the software behind it) is one process.

The distinction between a program and a process is a very interesting one, as it completely decouples some "executions" of a program from others and, as mentioned above, it allows to run one alongside the others without confusion between them. Each process has, among other things, its own status variables and data so that, if several instances of the same program are run at the same time (as were the terminals in the previous example), each one will keep its own history of commands (or actions), environment variables, etc. It is even possible (although not desirable) for one instance to hang, and this shouldn't keep the other from running.

One special kind of process is the one called 'job'. A 'job' is a process launched from a command interpreter. These processes are a little easier to handle than the others, but they have the most important features in common with regular processes.

As it runs, a process progresses through several possible states. The easiest one to understand is the "*running*" state. In it, the process is doing its work in an active manner, i.e., it is "*in execution*". If, for some reason, the process must wait (for instance, because it is waiting for some answer), the process can go into the "*suspended*" state. When the condition it waited for is fulfilled (for instance, when it gets the answer it was waiting for), it is ready to run again, so it goes into the "*ready*" state. A "*ready*" process need not be "*running*", since the processor may be running some other process (in principle, a processor can run just one process at a specific time; this is a simplification, since multi-core processors have, in essence, several processors; but, in this case, this discussion applies to each one of them). At any one time, some (possibly more than one) processes can be in the "*ready*" state. There is a part of the operating system (the *scheduler*) which decides which "*ready*" process can execute (in the case of multi-core processors, one process per processor). In fact, the scheduler can also decide to suspend a "*running*" process (even could keep on running) to let another process execute a little. There is a state which is similar to "*suspended*", and that is "*stopped*". "*Stopped*" processes are waiting to be allowed to keep on running. State transitions are executed by means of "*signals*", which are a means of communication between processes. They are explained later on, but, by way of introduction, with signals a user can terminate a process or, as mentioned, temporarily stop its execution.

There is one last state which we will consider. It is called the "*zombie*" state, and it is something more complex. It has to do with the fact that all processes must be launched by some other process. In the first section we saw how the command interpreter, when it gets a command from the user (for instance, the `ls` command), launches this command. What it does is create a new process (different from itself). The new process consist of the software that makes up the `ls` command. In a situation such as this, the command interpreter is said to be the "*parent*" process, and the `ls` command the "*child*" process. In becoming a "*child*" process, the `ls` command becomes a process itself, with all the features involved (memory, data, environment variables, code, etc). All processes in the system make up a hierarchy, a tree-like structure, where every process hangs from its "*parent*" process.

Going back to "zombie" processes, when a process terminates, it usually notifies its parent process, so it can have the operating system liberate the resources assigned to the child process. If the parent process terminates before the child process does, then it usually waits for the child process to terminate (or all of them, if it has more than one), before it does. That is, it doesn't really terminate until all its children have. However, it may happen that, for some reason, the parent process (in the example, the command interpreter) does not wait and terminates (for some system malfunction). In this moment, the child processes become disconnected from the process hierarchy in the system. The operating system cannot eliminate them from the list of processes nor free their resources. These are the "zombie" processes and, although not properly in the *running* state, since they have already finished, nor *suspended* or *stopped*, they still exist.

Input and output.

Standard input refers to the place from where a process gets the data it works on if no other indication is given. Analogously, standard output refers to where it places its results in the absence of more specific indications. Both standard input and standard output are usually assumed to be the terminal (the screen for output and the keyboard for input). For instance, the `ls` command places its listing in the standard output, that is, the screen. As another example, the `cat` command, which is used to concatenate files, works on whatever it reads from the keyboard if no file is specified as input.

All processes, including those that we create later on in the lab, must have a defined standard input and output. By default, these are the keyboard and the screen. However, other sources or destinations can be specified. For instance, one can keep the result of `ls` in a file by redirecting the command's output to that file when the command is invoked, so:

```
bash-ln.05$ ls > listing
```

In this case, "`listing`" is the file where the listing will be put into. The symbol ">" specifies the redirection of the output. For a redirected input, the corresponding symbol is "<". If the file for a redirected output already exists, the contents will be deleted. Instead, if one prefers the output to be appended to the content already present, then the symbol to use is ">>".

There is a second output stream for processes, and that is the "standard error". This is where a process can show its error messages, and it is different from the standard output, in case one prefers not to mix up the program results with the error messages. The standar error is also the terminal, but it can be independently redirected. In this case, we use the symbol "`2>`", like so:

```
bash-ln.05$ls 2> error
```

This would stream to file `error` any possible error messsage that `ls` could generate, but not the results of the command, that is, the listing of the working directory. 2 is the standard error "handle". The one for standard input is 0 and for standard output is 1. Thus, when we do:

```
bash-ln.05$ ls > listado.
```

in reality, what we are doing is:

```
bash-ln.05$ ls 1> listado.
```

What happens is, as this action is performed a lot more frequently than redirecting the standard error, the omision of the handle is interpreted to mean that the redirected stream is the standard output.

Concatenated processes ("pipes").

Sometimes it is necessary to execute a succession of commands where the output of one is the input of the next one. In this case, one can pipeline the execution of these commands so they will execute one after the

other, redirecting the output of one command to the input of the next one. This is done by creating a "pipe" structure:

```
bash-ln.05$ ls -la | grep "nothing"
```

The conection of the `ls` command and the `grep` command is created by the "`|`" symbol. The output of the `ls` command, which would normaly be shown on the standard output (the screen), is sent as input to the `grep` command, which would otherwise work on the keyboard. (The `grep` command is used to find patterns in its input; it is explained later on, but it is advisable to look at the manual page for it at this point). The net result is that `grep` looks for the word "`nothing`" in the listing generated by `ls`. Any number of commands can be concatenated in this way.

Background execution.

When using the terminal, it is usual for the user to key in the command to run so it will execute and interact with the user. This is called the "*foreground*" execution. Alternately, in order to be able to execute several commands simultaneously (not in sequence, as with a pipe, but at the same time) it is necessary for all of them but one to be able to run in what is called "*the background*", that is, without user interaction (at least by means of the terminal). For instance, if one of the commands to be run is a listing of all the files and directories in the system (`ls -Ra`), which would probably take a while, and at the same time check which users are logged onto the system (with the `who` command; look up the man page), one can generate the listing in the background, so that, while it is being generated, the terminal is free to accept another command. To do this:

```
bash-ln.05$ ls / -Ra >CompleteListing&
```

The "&" symbol at the end of the command line tells the command interpreter to run the command in the background, so the interpreter will launch it and return to the user. While a process is running in the background it shouldn't access neither the keyboard nor the screen, that is, it should not interact with the user, since that would cause the messages to merge with those of the process running in the foreground. Only those processes which don't need user interaction are good candidates to run in the background. If the process creates output, it must be redirected before it is launched, like this, for instance:

```
bash-ln.05$ ls > /dev/null&
```

The previous command runs in the background and generates a listing of the working directory, but puts it in file `s`, which is a special file which doesn't keep any information. Its use here guarantees that the command will execute successfully. The listing would not be shown on the screen, thus avoiding any interference there with the output from another command running in the foreground.

By being able to run commands in the background, a user can have several processes running in parallel. These processes would be the different "*jobs*" of the terminal they are launched from. It is possible to see which ones these are with the `jobs` command (check the `man` page). Each job has an identifier (this is a different thing from the process identifier, PID, which we will talk about later) and it can be used to, among other things, move it to and from the background and the foreground. There are two commands for this. The command `bg` sends a job to the background, and the command `fg` brings a job to the foreground.

Search command ("grep").

When one works with big amounts of information (like listings, for instance), it is sometimes interesting to filter out whatever is not essential to the task at hand. For instance, file `/etc/passwd` has a list of all users in the system and, for each one, it contains some characteristics. One of them is the default command interpreter for the user, since it is possible to choose from a variety of them in Linux. The following command:

```
bash-ln.05$grep bash /etc/passwd
```

will look in the lines of `/etc/passwd` which of them contains the string "`bash`", which is one of the command interpreters available, and will show them (the whole line). This will let us know all the users who use `bash` by default. `grep` is a very versatile command which allows to run searches more complex than just finding a simple pattern. For these complex searches one uses "*regular expressions*", which are formal methods to specify what must be looked for. For instance, one can look for lines which don't contain a specific pattern, lines where the pattern appears at the beginning of the line, or at the end, etc. It is advisable to look into the man page for `grep` to learn some of its options and how to specify regular expressions, since it is a very useful tool.

Process management.

In Linux, the work that a user performs is organized around processes. Each command (almost every action) that a users executes is a process, and that is why it is important to know the tools that Linux provides to manage processes.

The first of these tools is the `ps` command. `ps` provides information about a user's terminal processes. A process is labeled "*terminal*" when it was launched from the terminal. However, by means of flags, `ps` can be told to show other kinds of processes, or even processes from other users. The information that `ps` can display about a process is plentiful, and the most important is: the process ID (PID), the process name, the parent process ID (PPID) and the ID of the user who launched the process. For instance, the command:

```
bash-ln.05$ps -la
```

will generate a full listing ('`l`') of terminal processes from any user ('`a`') with the following columns:

```
UID   PID  PPID  F  CPU PRI  NI  SZ  RSS  WCHAN  S  ADDR TTY  TIME CMD
```

The command will continue to add one line each for every process found. Of all the columns, the most important are UID (the user's ID), PID (the process ID), PPID (the ID of the process's parent) and CMD (the command which generated the process). A listing is full when it has all the pertinent information, that is, all the columns. A regular listing has fewer columns. Compare the result of the command above with what the following one gets:

```
bash-ln.05$ps -a
```

Another command related to process information is "`top`". This command provides data about processes running in a system, with one line for each process, and sorts the lines as it is told. For instance, it can order by PID, by percentage of CPU usage, by process state, etc. An interesting feature of this command is that it dynamically updates the information as long as it is not terminated, do the information displayed is always up to date.

Another command worth mentioning is "`pstree`", which shows the hierarchical process tree, so one can see which children a parent process has.

SIGNALS

The `ps` or the `top` commands are useful for, among other things, getting the PID of a process, since processes are managed by means of this identifier. For instance, if a process hangs, that is, it stops responding, it can be terminated by means of a terminate signal. Signals are a means of communication between processes which can be used in a case like that. Processes are prepared to send and receive signals (when a program is written by the user, they must include this functionality in them, otherwise it won't work). There are many kinds of signals, and each one is identified by an integer number. The most widely used are `SIGINT` (number 2), which is like typing CTRL-C, that is, terminating the program, `SIGTERM` (number 15) which tells a process to terminate, but gives it the opportunity of doing so in an orderly manner, or `KILL` (number 9), which terminates it unconditionally and without a warning (useful when a process is not responding). The command to send signals to a process is `kill`:

```
bash-ln.05$ kill -15 21345
```

The command above will warn process with PID 21345 to finish. If the process is correctly programmed, it will store the data and terminate. If it hangs, it may be necessary to execute:

```
bash-ln.05$ kill -9 21345
```

which will terminate it without the option to do so in an orderly manner. In the commands above, instead of the signal number, it is possible to use the signal's name. For instance, in the first one, number 15 can be substituted by `TERM`:

```
bash-ln.05$ kill -TERM 21345
```

as 15 is the number which corresponds to `SIGTERM`.

As mentioned above, signals are a means of inter-process communication. To use it, programs must be created with the according functionality, that is, they must contain routines to deal with the signals. Otherwise, they won't notice them. Only one of them, `SIGKILL` (-9) cannot be handled by the program itself, since this signal completely terminates the process.

EXERCISES

*For every exercise below it is **again** advisable to look at the necessary man pages.*

1. Use the `cat` command to print in a file the information typed in the terminal with the keyboard (use `Ctrl+D` to stop keyboard input). It will be necessary to redirect the output towards the file using ">".

2. Repeat Ex. 1 in such a way that the new information does no erase the information generated in Ex.1

3. Repeat Ex.1 with a different file.

4. Show both files on the screen using just one command.

5. Show the contents of a directory in such a way that just directories (not files) are shown. It will be necessary to use a pipe with `ls` and `grep`.

6. Repeat Ex.5 but showing just the files whose owner is the standard laboratory user (larq).

7. Repeat Ex.5 but showing just the files that have been created in January (or any other month of your liking).

8. Do the following tasks:
    a. Use `cat` to create a process that prints information typed on the keyboard in the file `/dev/null`
    b. Obtain its PID (it could be necessary to open another terminal).
    c. Stop it with the appropriate signal (`SIGSTOP`).
    d. Check the state of the process.
    e. Obtain the process tree and find the process in it.
    f. Make it finish in an orderly fashion.

9. Obtain a list of all system processes that have been initiated by the administrator user *root*.