Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

ƒSO

Lab session 3
Process monitoring in LINUX

Content

1. Goals

The aim of this lab session is to be able to observe processes behavior in the system (i.e. CPU utilization, memory occupancy, PIDs, etc).
In order to achieve this, you will acquire skills about using the following tools:
- Process management (`ps` and `top`) and signalization (`kill`) commands.
- File system `/proc` where Linux collects all the process monitoring information.
- Tools to automate monitoring reports and process control:
    - Text filtering commands (`grep` and `tr`).
    - Shell programming.
    - `awk` programming.

2. Utilities and commands shell for process

Using command mkdir create a new directory named fso_pract3 to put all your lab files:

```
$ mkdir fso_pract3
$ cd fso_pract3
```

2.1. `ps` command

To monitor processes on Linux `ps` command is available.

```
$ ps u
    USER      PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
   gandreu    251   0,0  0,0  2435492   1112 s000  S     4:20PM   0:00.01 -bash
```

This command displays the list of processes in the system and is based on information stored in the file system `/proc`. Depending on the parameters of `ps` command it will show us more or less information. It has a lot of options and depending on them will display several columns of process information, like:

PID	Process id
PPID	Parent process id
UID	Process user owner id
TTY	Terminal associated to the process. If there is none it appears ?
TIME	Cumulated CPU time taken by the process
CMD	Program or command name that has initiated the process
RSS	Memory size (in Kbytes) of the process resident part
SIZE	Virtual size of the process image
NI	Process priority value (nice). Less means more priority
PCPU	Percentage of CPU used by the process
STIME	Process starting time
STAT	Process state

Process states that appear in the STAT column can be:
- R (Runnable): In execution.
- S (Sleeping): Process waiting for some event to continue.
- T (sTopped): Process completely stopped. It can be restarted.
- Z (Zombie): Zombie process.
- D (uninterruptible sleep): Process in uninterruptable waiting that they are associated to I/O actions.

Along with the letter indicating the status of the process, additional characters may appear that give more information about the process status. The most common are:

- +: The process is running in the foreground.
- X : The process being debugged or traced.

By default, `ps` command displays information regarding PID, TTY, TIME and CMD. Running it with different options we can control how much information is displayed.

1.Exercise:
Execute the following commands and observed the differences in their output.

```
$ps
$ps u
$ps -la
$ps f
$ps aux
```

`ps` is usually used in conjunction with `grep` command, explained later.

2.2. `top` command

`top` command shows in real time a list of processes currently running on the system, specifying the % of CPU and memory consumed, their PIDs, owner users, and so on. Execute the top command and the terminal will display information divided in two parts, like this:

```
Processes: 72 total, 2 running, 1 stuck, 69 sleeping, 322 threads                              16:58:05
Load Avg: 0.15, 0.17, 0.16  CPU usage: 2.87% user, 3.34% sys, 93.77% idle  SharedLibs: 16M resident, 11M data, 0B linkedit.
MemRegions: 11390 total, 385M resident, 39M private, 431M shared.
PhysMem: 627M wired, 936M active, 193M inactive, 1756M used, 2340M free.
VM: 164G vsize, 1123M framework vsize, 84839(0) pageins, 0(0) pageouts. Networks: packets: 7535/6626K in, 5609/1252K out.
Disks: 22675/1057M read, 12667/355M written.

PID  COMMAND     %CPU TIME     #TH #WQ #POR #MREG RPRVT  RSHRD  RSIZE  VPRVT VSIZE  PGRP PPID STATE    UID FAULTS  COW
288  Grab        3.8  00:01.72 5   4   108- 211+  4088K+ 40M-   15M+   29M+  2483M+ 288  143  sleeping 504 13620+  366
286  quicklookd  0.0  00:00.05 4   1   72   73     2468K  7384K  6016K  534M  2920M  286  143  sleeping 504 1921    228
281- CVMCompiler 0.0  00:00.03 1   0   27   38     904K   460K   1892K  18M   591M   281  143  sleeping 504 622     81
280  xpchelper   0.0  00:00.02 2   2   39   41     972K   220K   4468K  30M   2390M  280  1    sleeping 504 1401    168
278  mdworker    0.0  00:00.12 3   1   55   63     1392K  9564K  5584K  23M   2411M  278  142  sleeping 89  2424    202
```

The top output on the screen shows among other things, the system uptime, number of connected users, average load, the number of processes running on the system, as well as disk, memory and cpu(s) utilization. Then we can see a list of processes that can be sorted by CPU usage or memory, which is an excellent aid to detect processes that consume excessive resources. This list shows several attributes of every process, such as the process PID, the user running it, %CPU and memory space consumed, command associated to the process, process execution time, and others.

¡warning! To exit top just type 'q' key.

2.3. `kill` command

`kill` command serves not only to kill or terminate processes, but mainly to send signals to processes. The default signal (when none is indicated) is to finish or kill the process, and the syntax is:

$ kill SIGNAL PID

Where SIGNAL is the signal to send and PID is the process ID to which the signal SIGNAL is delivered.
Get a list of all the signals of your machine by executing:

$ kill -l

```
ivanovic:~ gandreu$ kill -l
 1) SIGHUP        2) SIGINT        3) SIGQUIT       4) SIGILL
 5) SIGTRAP       6) SIGABRT       7) SIGEMT        8) SIGFPE
 9) SIGKILL      10) SIGBUS       11) SIGSEGV      12) SIGSYS
13) SIGPIPE      14) SIGALRM      15) SIGTERM      16) SIGURG
17) SIGSTOP      18) SIGTSTP      19) SIGCONT      20) SIGCHLD
21) SIGTTIN      22) SIGTTOU      23) SIGIO        24) SIGXCPU
25) SIGXFSZ      26) SIGVTALRM    27) SIGPROF      28) SIGWINCH
29) SIGINFO      30) SIGUSR1      31) SIGUSR2
```

Signal SIGKILL (9) does process termination; this signal cannot be neither handled, masked or ignored.

2.Exercise:

Launch the `yes` command and then end up the process by sending it signal SIGINT typing in the keyboard Ctrl-c

```
$ yes
y
y
  ctrl-c
```

Now launch the command `yes` in the background, and find its PID, then finish the process using `kill`, after this check with `ps` that the process no longer exists.

```
$ yes >/dev/null &
$ ps -la
$ kill -SIGKILL PID
```

The comand kill permit to reference the signals by its number or name, example:
```
#> kill -9 11428         (ends process with PID 11428)
#> kill -SIGKILL 11428  (does the same as the former command)
```

`killall` is similar to kill with SIGKILL signal but instead of the PID it is specified the program name. `killall` ends all processes with the specified program name.

3.Exercise:

 Check `killall` command using following command sequence:

```
$ yes >/dev/null &
$ yes >/dev/null &
$ yes >/dev/null &
$ ps -la
….
$ killall  yes
$ ps -la
```

4.Exercise:

Launch Kate editor from a terminal in background. In the same terminal, run "ps" command to find the started process PID. Use the kill command to "kill" the Kate process.

3. "/proc" directory

This directory maintains information about the processes currently running. By processing the information in `/proc` we can get the same data provided by commands like `ps` or `top`.

Note. On MacOS there is no /proc, system info is stored in another place. Please, look at the annex.

In `/proc` directory there will be a sub-directory `/proc/$pid` for each running process and a set of files with system information:

- **cpuinfo**: where lots of information can be found about the processor.
- **stat**: it contains information about the processor performance, it includes information about the time the processor has been on, the number of running processes, etc.
- **version**: it contains information about the kernel version, gcc compiler, Linux distribution, etc.

5.Exercise:

Run the following script to check the contents of /proc, also check that there is a directory whose name is the pid of your shell, to find this pid run the ps command (replace $pid_bash for the PID of your shell):

```
$ ps
……•
$ ls /proc
…
$ ls /proc/$pid_bash

$ more /proc/cpuinfo
```

Within each process folder (/proc/$pid) we will find several files. Some of them are:

- **/proc/$pid/cmdline**: it contains information about the name of the process, including its full path.
- **/proc/$pid/stat**: it contains much information about the process. It is a text file made up of several fields, which are separated by the space character. Among the several fields available we have the PID, process name (just the name of the process, not the full path, etc.).
- **/proc/$pid/maps:** it contains information about process memory map.
- **/proc/$pid/status:** It contains information related to a particular process like its PID, PPID, state and attributes related to memory consumption.

```
gandreu@shell-sisop:~$ more /proc/$$/status
Name:    bash
State:   S (sleeping)
Tgid:    10956
Pid:     10956
PPid:    10955
TracerPid:      0
Uid:     1692409199      1692409199      1692409199      1692409199
Gid:     1692452651      1692452651      1692452651      1692452651
FDSize: 256
Groups: 1692441063 1692452651 1692453521 1692453528 1692459925 1692459927 169247
8937 1692515549
VmPeak:    21168 kB
VmSize:    21168 kB
VmLck:         0 kB
VmHWM:      4108 kB
VmRSS:      4108 kB
VmData:     2672 kB
VmStk:        88 kB
VmExe:       760 kB
VmLib:      1920 kB
VmPTE:        56 kB
Threads:       1
SigQ:    0/8191
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000010000
SigIgn: 0000000000384004
SigCgt: 000000004b813efb
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
Cpus_allowed:   03
Mems_allowed:   00000000,00000001
voluntary_ctxt_switches:        88
nonvoluntary_ctxt_switches:     59
gandreu@shell-sisop:~$
```

The previous screen shows the "status" file content:
- ▪ **VmSize**: Total virtual space demanded by the process.

- **VmLck**: Space taken by pages that the process has blocked in its virtual space. This blocking mechanism avoids these pages to be selected as victims by the page replacement algorithm.
- **VmRSS**: It gives the "resident set size", that is, the size of the page set actually loaded in memory. In other words, it indicates how much physical memory is consuming the process.
- **VmData**: It indicates the virtual size assigned to the process data.
- **VmStk**: It indicates the virtual size assigned to the process stack.
- **VmExe**: It indicates the virtual size assigned to the program code that executes the process.
- **VmLib**: It indicates the virtual size assigned to the dynamic libraries being used by the process.

Full and comprehensive information about these files and others can be found using command:

```
$ man proc
```

4. Tools to automate process monitoring

4.1. grep command

grep command searches for a regular expression in a file. Can be used alone to locate strings in files or standard input redirected through a pipe. For instance:

```
$grep hello file       //shows the lines that have the string "hello" in the file "file"
$grep -i Hello file   // shows the lines that have "hello" in the file "file" ignoring case
$ls-l|grep pract3 //shows the lines  pract3 in the input stream produced by the output of ls -l
```

6.Exercise:
Write a command line that shows on the screen the clock frecuency of the processors in the system. You have to access file /proc/cpuinfo and to use grep command to select the appropriate line.

7.Exercise:
Write a command line that shows on the screen the cache memory size of the processors in the system. You have to access file /proc/cpuinfo and to use grep command to select the appropriate line.

4.2. Shell script

The LINUX shell (i.e. bash) is an executable program (like any other user program) which is available on any Linux system and whose purpose is to read commands (text lines) that specify program executions, analyze them and make the system calls that are necessary to run them.

This interpreter defines, as with any other interpreter or translator, a programming language that has features like:
- Procedures, better known as shell scripts
- Words and reserved characters (also known as wildcards)
- Variables
- Structures for flow control such as if, while, etc.
- Interrupt handling

Actually in Linux there are several shells, every user can choose anyone of them. The most basic shell is Bourne shell, or sh, which is available on any system. Other common shells are ksh (Korn shell), csh (C like syntax shell) and bash (Bourne-Again Shell) that is fully compatible with sh while incorporating features from ksh and csh.

This lab session will introduce the features and syntax of sh, which are also applicable to ksh and bash.

Commands to be execute bye the shell can be provided via the keyboard or by a program contained in a file. In the later case the file is called a shell script. It contains commands to be executed by the shell, line by line. For example, you can create a file with the following content:

```
#!/bin/bash
# content
Num_process=$(ls –d /proc/[1-9]*|wc –l)
echo Number of System process is: $Num_process
```

Lines beginning with a # character are comments. The rest are statements that will be executed by the shell. In particular, echo command shows on the console the specified string. The "ls -d" command is used to obtain a list of files directory, while "wc –l" compute the number line.

8.Exercise:
Create the previous file using a text editor (i.e. Kate) and name it "content". You must to give execution permission to content to execute:

```
$ chmod +x content
$ ./content
```

Notice that the shell script file name used in the latter method "./content", not just "content". It has been indicated that the file to run is in the current working directory (directory "."). To avoid having to specify the path of an executable file when it is started to run, we can use environment variable PATH (see annex 1).

A shell script can be invoked with arguments which can be referenced as $0, $1, $2, $3, etc ... The reference $0 points to the program name, while $@ references all the arguments and $# is the number of arguments (both $@ and $# don't include $0).

9.Exercise:
Create a file named "arguments" with the following content:

```
#!/bin/bash
# arguments
echo The number of arguments is: $#
echo The full command line entered is: $0 $@
echo The command name is: $0
echo The first argument is: $1
echo The second argument is: $2
echo The third argument is: $3
```

Then give to file "arguments" execution permission and execute it with different parameters, like this:

```
$arguments one two three

$ arguments  FSO TCO ESO
```

4.2.1. Variables
A variable is created using "=" to assign to its identifier a value or content, this is, identifier=content. You access to a variable content using $ before the variable identifier. For instance:

```
$  name=Alberto
$ subject=FSO
$ msg="Hello World"
```

```
$
$ echo $name
    Alberto
$ echo $subject
  FSO
$ echo $msg
 Hello World
```

Warning: If you leave spaces between "=" and the identifier and/or the value then the shell will take it as if they were shell commands instead of a variable assignment.

4.2.2. for loops

The `for` statement is controlled by an associated variable that iterates over a list of values. Its syntax is the following:

```
for variable in value list do
sentencias
done
```

We can use a `for` loop to list the arguments in a command:

```
#!/bin/bash
echo The number of arguments is: $#
echo The entered command line is: $0 $@
echo The command arguments are:
for i in "$@"
do
   echo $i
done
```

The list of values in a `for` loop can also be the current directory file list. For example, the following program creates a backup of every file in the current directory:

```
for k in *
do
   cp $k $k.bak
   echo $k copy created
done
```

Finally, the list of values in a `for` loop can also come from a command execution using `$()`. For example:

```
for i in $(ls)
do
   echo $i
done
```

4.2.3. if statement
The `if` statement allows conditional execution of commands. Its syntax is the following:

```
if command executes ok then
   command block
else
```

```
   alternative command block
fi
```

Notice that the condition in the if statement is not an expression but a command. The condition is true if the command "completes successfully" (in this case the statements following "then" are executed) and false otherwise (in this case the statements following "else" are executed). The "else" clause is optional.

4.2.4. test command

The `test` command allows evaluating conditions and, therefore, is useful to use it in conjunction with the if sentence. The types of expressions that it can evaluate are:

Numerical expressions. The general form of numerical expressions is:

N <primitive> M

where N and M are interpreted as numerical values. The primitives that can be used are:

-eq	True if N y M are equal
-ne	True if N y M are not equal
-gt	True if N is greater than M
-lt	True if N is less than M
-ge	True if N is equal or greater than M
-le	True if N is less or equal than M

10. Exercise:

Edit a file named "my_processes" with the following commands and execute it:

```
#!/bin/bash
# my_processes
process=$(ps u | grep $USER | wc -l)
if test $process -gt $1
then
echo "More than $1 user processes active"
else
echo "Equal or less than $1 user processes active"
fi
```

Alphanumeric expressions. Let S and R be strings. We can have two types of expressions:

<primitive> S
S <primitive> R

The primitives that can be used are:

S=R	S and R are the same
S != R	S and R are not the same
-z S	True if S is empty
-n S	True if S in not empty

File type. The general form of the expressions is:

<primitive> file

The primitives that can be used are:

-s	True if the file exists and it is not empty
-f	True if the file exists and it is a regular file (not a directory)
-d	True if the file is a directory
-r	True if the file has reading permission
-w	True if the file has writing permission

-x	True if the file has execution permission

11.Exercise: *Implement a script **del_file**, with one argument that will be a file name. The script has to check if the file exists and that it isn't a directory. If it doesn't exist a message on the screen will notice it, else if it is a regular file it will be deleted. In case of being a directory a message on the screen will tell that it is a directory and that it can't be deleted.*

4.3.   awk

awk is a tool that is useful to modify files, search and transform databases, it also creates simple reports. It is a programming language then awk has many capabilities, but this section is intended to illustrate its basic use.

The basic function of awk is to find lines in files or other text units that contain certain patterns. When a pattern is found in a line, awk performs the action specified for that pattern on that line. awk continues processing input lines in this way until it reaches the end of file.

4.3.1. Syntax
awk can be invoked in different ways, but in any case we must indicate what patterns it has to look for, what actions it has to perform on the lines containing those patterns and from where it takes the input lines. Patterns and actions are indicated by a program.

```
$ awk 'awk_program' input_file_1 input_file_2 ...
```

This format tells the shell to run "program" using awk to process the lines in the input files "input_file_1" and "input_file_ 2".

When the program is long it is more convenient to put the program into a file and execute it as follows:

```
$ awk –f prog-file fichero_de_entrada_1 fichero_de_entrada_2 ...
```

In this case "prog-file" is the file that contains the program.

An awk program consists of a set of rules. Each rule specifies a pattern to search, and an action to perform when that pattern is in found. The action is enclosed in braces to separate it from the patterns.

```
pattern { action }
pattern { action }
...
pattern { action }
```

4.3.2. Patterns
Patterns can be specified by regular expression enclosed in slashes /. The simplest regular expression is a sequence of letters, numbers or both. For example, the regular expression root matches any string containing "root". Therefore, the pattern /root/ matches any record containing the string "root".

4.3.3. Actions

An action causes something to happen when a pattern is found. If no action is specified, awk {print} is applied (default action). This action copies the record (usually a line) of the input file to the standard output. The print statement can be followed by arguments by forcing awk to print only the arguments.
The following examples show the use of awk.

If file test.txt is generated as follows:

```
echo -e "Column1\tColumn2\tColumn3\tColumn4\n" > example.txt
```

Then the following cat command:

```
cat example.txt
```

will return on the console the following output:

```
Column1         Column2         Column3         Column4
```

If you use awk to only display column 1 and column 3 as follows:

```
awk '{ print $1, $3}' example.txt
```

The output will be:

```
Column1 Column3
```

If you add data to the file as follows:

```
echo -e "Data1\tData2\tData3\tData4\n" >> example.txt
echo -e "Data5\tData6\tData7\tData8\n" >> example.txt
echo -e "Data9\tData10\tData11\tData4\n" >> example.txt
```

Then the following cat command:

```
cat example.txt
```

will give:

```
Column1         Column2         Column3         Column4
Data1   Data2   Data3   Data4
Data5   Data6   Data7   Data8
Data9   Data10  Data11  Data4
```

If you use awk again to only display column3 and column1 (in that order):

```
awk '{ print $3, $1}' example.txt
```

the output will be:

```
Column3 Column1
Data3 Data1
Data7 Data5
Data11 Data9
```

You can use awk to display only the line that contains the regular expression /Data5/ , with the following command:

```
awk '/Data5/ { print }' example.txt
```

The output will be:

```
Data5   Data6   Data7   Data8
```

If you want to output only columns 1 and 4 then:

```
awk '/Data5/ { print $1, $4}' example.txt
```

will give:

```
Data5 Data8
```

12.Exercise:
Looking into the file /etc/passwd the "root" string, and print out the line where the string is found

 13.Exercise:

Implement a Shell script named "inf_process" that takes as argument a process PID and then it prints on the terminal in column format the given process PID, PPID, STATE and COMMAND. Remember that this information can be found in files `/proc/$pid/status` and `/proc/$pid/cmdline`. The output will be like the following:

```
        PID   PPID      STATE   COMMAND
        8900  8880        S     /bin/bash
```

14.Exercise:

Implement a Shell script named "system_processes" that will deliver to script "inf_process" all the running process PIDs in order to print their PID, PPID, STATE and COMMAND.

4.3.4. Splitting the input into records and fields

The input to a typical awk program is read in units called records, and they are processed by the rules one by one. By default, a record is a line in the input file.

The awk language divides its input records into fields. By default, fields are separated by spaces. The way awk splits records into fields is controlled by the field separator, which is a single character or a regular expression. awk analyses the input record looking for field separator matches, text fields are the text found between consecutive separators. For example, if the field separator is ':', then the following line:

patricia:x:1000:1000:Patricia Balbastre Betoret:/home/patricia:/bin/bash

will be partitioned into 7 fields:

'patricia', 'x', '1000', '1000', Patricia Balbastre Betoret', '/home/patricia' and '/bin/bash'.

The field separator can be specified in the command line using the argument '-F'. For example:

```
$ awk –F: 'program' input_files
```

Set as field separator the character ':'.

To refer to a field in an awk program, using the $ symbol followed by the number of the field. So $1 refers to the first field, $2 the second and so on. $0 is a special case, it represents the entire input record.
Another example, if we write:

```
$ awk –F: '/model name/ {print $2}' /proc/cpuinfo
```

It would print the second field separated by ':' lines containing "model name" in the file "/proc/cpuinfo".

Optional Exercise:

Start 5 `Kate` processes, some of them in the foreground and the others in the background. Make a script, looking for information in `/proc` to show in the screen the number of Kate processes running, additionally for every process it will show its PID, its status and its memory use. To achieve this you should use the knowledge acquired in the previous sections.

5. Annex

5.1. Annex 1: PATH variable

To avoid having to specify the path of an executable file when it is started to run, we can use environment variable PATH. This variable contains the list of directory names (separated by the character ":") where the shell searches for executable files.

We can see the value of PATH with the following command:

```
$ echo $PATH
```

Check if the value of the PATH environment variable, that the shell uses, contains the current working directory. If so check that the result of executing the file "content" is the same using with or without the prefix "./".

```
$contenido
```

If it is not the case, the local working directory can be appended to PATH with the following command:

```
$ PATH=$PATH:.
```

This change will only last until the end of the active shell instance (i.e. until closing the terminal). If you want this change to be permanent then you should edit the configuration file $HOME/.bashrc and append at the end the previous command.

Exercise:
Edit as described previously file $HOME/.bashrc, then open a new terminal and verify the new persistent PATH VALUE with:

```
$xemacs $HOME/.bashrc

$ echo $PATH
```

5.2. Annex 2: Arithmetic operations

The shell can evaluate an arithmetic expression:

```
$((expression))
     evaluates the arithmetic expression and replaces it by the result
```

Example:

```
$ echo 1+1
1+1
$ echo $((1+1))
2
```

Some arithmetic operators supported are:

```
+     addition
*     mutiplication
-     substraction
/     integer division
%     integer division remainder
( )   operation grouping
```

5.3. Annex 3: while loops

This is another loop structure that can run a block of commands while a condition evaluates to true:

```
while CONDITION; do
      command block
done
```

At the beginning of every iteration the condition is evaluated and then the loop ends if it is not true.

while loop examples:

```
# while loop controlled by an integer variable (equivalent to seq 1 5)
i=1
while [ $i -lt 6 ]; do
      echo $i
      i=$(($i+1))
done

# reads a line from the standard input until the line be 'quit'
read line
while [ "$line" != "quit" ]; do
  read line
done
```

5.4. Annex 4: awk patterns BEGIN and END

BEGIN and END are special patterns. They are used to supply to awk what to do before it starts processing and after it ends processing the input records. A BEGIN rule is executed once before reading the first input record. The END rule is executed once after awk has read all the input records. For example:

The following command specifies to start printing the phrase "Hello World" and the finish.

```
awk 'BEGIN { print "Hola mundo"; exit }'
```

This will return an output like this:

```
Hola mundo
```

Another example:

```
$ awk 'BEGIN {print "How many times appears Data4" ; data=0 ; }
>/Data4/ {++data}
>END {print "Data4 appears "data" times"}' example.txt
```

This program finds how many times appears Data4 in the file generated in the previous examples "example.txt". The BEGIN rule prints a title for the report and initializes the counter data to 0, this initialization is not required since awk does this by default.

The second rule increments the variable "data" every time awk reads in the input a record that contains pattern /Data4/. The END rule prints the final value of "data" variable.

5.5. Annex 5: awk `IF` statement

The if-else is a sentence for decision making in awk. It looks like this:

if (condition) then-body [else else-body]

where condition is an expression that controls what does the rest of the sentence. If the condition is true, then it executes the "body-then" sentences, otherwise else-body sentences are executed (assuming that the else clause is present). The else part of the sentence is optional. The condition is considered false if its value is 0 or a null string, otherwise it will be considered true.

Here there is an example:

```
$ awk '{ if ($0 % 2 == 0)
> print $0 "it is even"
> else
> print $0 "it is odd" }'
```

In this example, if expression "$0 % 2 == 0" is true (i.e. the value passed to it is divisible by 2), then it executes the first print statement, otherwise it executes the second print statement.

If "else" appears in the same line as "then-body" and the "then-body" is a compound statement (not between braces), then a semicolon must separate the "then-body" from the "else-body". To illustrate this consider the following example.

```
$ awk '{ if ($0 % 2 == 0) print $0 "it is even"; else print $0 "x is odd"}'
```

If you forget to put ";" it will cause a syntax error. It is recommended not to enter the else statement like this, because it may lead to confusion.

Exercise:
Use what you have learned so far to make a script that kills the process Kate with lower PID among several Kate processes that have been previously started.

5.6. Annex 6: `tr` utility

This command allows changing or translating the characters from the input according to rules specified. Its general format is:

```
$ tr [options] string_1 string_2
```

Some examples using this command are:

- *To change a character by another.*
  Example: replace the character used as separator between fields (':') to tab character.

  ```
  $ tr ':' '\t' < input_file
  ```

  Example: replace null characters by spaces.

  ```
  $ tr "\0" " " < input_file
  ```

- *To change a set of characters.*

Example: capitalize all characters that appear in a file:

```
$ tr '[a-z]' '[A-Z]' < input_file
```

Exercise:
Replace string Data4 by Data5 in file example.txt.

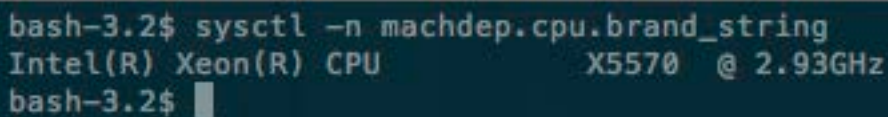## 5.7. Annex 7: Getting system info in Mac OSX

In Mac OSC, appart from apps "System information" and "Activity monitor" that deliver graphic information in real time about processes and system behaviour, you can also get information by means of the command line.

In Linux CPU information is located in file `/proc/cpuinfo` .For instance to get the CPU type you do:

```
$ cat /proc/cpuinfo | grep "model name"
```

In OSX, you use **sysctl** command to get information from several sides of the OSX kernel states. For instance, you get the OSX name by doing:

```
$sysctl -n machdep.cpu.brand_string
```

```
bash-3.2$ sysctl -n machdep.cpu.brand_string
Intel(R) Xeon(R) CPU           X5570  @ 2.93GHz
bash-3.2$
```

And in order to get complete CPU information you do:

```
$sysctl -a | grep machdep.cpu
```

To get the number of cores and threads you can do:

```
$sysctl -a | grep machdep.cpu | grep core_count
$sysctl -a | grep machdep.cpu | grep thread_count
```