Presentation

This practice proposes a set of activities focused on the aplication of some of the concepts introduced in the first modules of the subject on a Unix system.

The student will have to do differents experiments and answer the questions proposed. You will also have to write a small program in C language.

The practice can be developed on any Unix system (the UOC provide you the Ubuntu 14.04 distribution**). It's advised that while performing the experiments there are no other users working on the system because the result of some experiments may depend on the system load.**

Each question suggests a possible timing to be able to finish the practice before the deadline and the question weight in the final assessment of the practice.

Competencies

Transversals:

- Skill to adapt to technologies and future environments by updating professional skills

Specifics:

- Skills to analyze a problem at the level of abstraction appropriate for each situation and apply the skills and knowledge acquired to address and
- Skills to design and build computer applications using development, integration and reuse techniques.

Practice proposal

In order practice some contents, we provide you with the pr1so.zip file and source files. Unzip it with the unzip pr1so.zip command. To compile a file, for example prog.c, you need to run the command gcc -o prog prog.c1.

Modules 1 and 2 [From March 10 to 24] (40% = 5% + 15% + 20%)

We provide you the benchmark.c and benchmark lib.c programs and the test.sh i test lib.sh shellscripts (you don't need to analyze how they are implemented).

• benchmark.c is a program with a single execution thread that copies a source file over a destination file. The program is parameterized with the size of the intermediate vector (buffer) used to make the copy. In order to copy a file you need to make a loop where at each iteration the call to the system is invoked that reads from the source file as many characters as the size of the buffer and they are temporarily stored in the buffer; then the contents of the buffer are written to the destination file. For example, if the buffer size is 4 characters, in the first iteration they will be read/written the first 4 characters of the file, in the second the characters

from the fifth to the eighth, and so on. Consequently, the larger this buffer is, the less iterations the loop will need to process the source file.

• test.sh is a script that sequentially launches different executions of the benchmark program on the same source file. The difference is the size of the buffer (32768, 16384, 8192, ..., 4, 2, and 1 characters) and, in each case, shows (in seconds) the execution time (elapsed, wall time), the time consumed in user mode and in system (privileged) mode.

• benchmark lib.c is similar to benchmark.c but, instead of using System calls to read/write the files, it uses library routines. It also reads/writes to files using an intermediate buffer of the specified size.

• test lib.sh is similar to test.sh because it invokes benchmark lib.c instead of benchmark.c.

You can see an exemple below

```
[enricm@pcmartorell-i-ribas codi]$  ./test.sh
Testing size 32768 :    0.01 elapsed,   0.00 user,      0.01 sys
Testing size 16384 :    0.02 elapsed,   0.00 user,      0.02 sys
Testing size  8192 :    0.02 elapsed,   0.00 user,      0.01 sys
Testing size  4096 :    0.02 elapsed,   0.00 user,      0.02 sys
Testing size  2048 :    0.04 elapsed,   0.01 user,      0.02 sys
Testing size  1024 :    0.07 elapsed,   0.02 user,      0.04 sys
Testing size   512 :    0.12 elapsed,   0.02 user,      0.10 sys
Testing size   256 :    0.24 elapsed,   0.05 user,      0.18 sys
Testing size   128 :    0.47 elapsed,   0.14 user,      0.32 sys
Testing size    64 :    0.94 elapsed,   0.25 user,      0.67 sys
Testing size    32 :    1.85 elapsed,   0.58 user,      1.26 sys
Testing size    16 :    3.62 elapsed,   1.11 user,      2.50 sys
Testing size     8 :    7.33 elapsed,   2.20 user,      5.06 sys
Testing size     4 :    14.58 elapsed,  4.62 user,      9.95 sys
Testing size     2 :    29.05 elapsed,  9.19 user,      19.80 sys
Testing size     1 :    57.94 elapsed,  18.26 user,     39.66 sys
[enricm@pcmartorell-i-ribas codi]$
```

Compile benchmark.c and check that test.sh works on your system in the same way to the example (the same amount of lines should appear but the specific runtime values will be different although they should follow the same trends).

Answer the following questions justifiably

1.1. Hardware analysis:

1.1.1. Please indicate which specific processor model you are working on. To do this, you can proceed as in this example:

```
[enricm@willy dev]$  grep "model name" /proc/cpuinfo | head -1
model name      : Intel(R) Core(TM) i5 CPU         650  @ 3.20GHz
[enricm@willy dev]$
```

1.1.2. How many physical cores (cores) does your processor have? If your processor is Intel, you can check http://ark.intel.com/.

1.1.3. How many cores are you really using? This number may be different from the one obtained in 1.1.2. depending on how the operating system is configured, the virtual machine (if you are using one) or if you have Hyperthreading enabled. To answer this question, count how many lines the following command writes and attach a screenshot with the result obtained.

```
grep processor /proc/cpuinfo
```

If the result is lower than the number obtained in 1.1.2., reconfigure your system/virtual machine so that the number of cores used is at least equal to the number of available cores. Please indicate how you have reconfigured the system and show the result of grep processor /proc/cpuinfo again.

1.2. Top command execution:

The top command displays information about running processes on the machine. From a window launch top.

Answer the following questions justifiably. **Attach screenshots with the commands running results.**

1.2.1. Explain the information displayed by the first three lines of the top result. (You can consult the man top system manual).

1.2.2. From another window (and without closing the window where top is running), run the test.sh script. Notice the changes that appear in the third line shown by top (the one for percentages) while the script is running. Justify them and relate them to your answer to 1.1.3.. Focus on the percentages us, sy and id.

1.3. Shellscripts results analisys:

1.3.1. Notice the execution times shown by the test.sh script for the various tests. Try to explain the observed trends. Remember that in all the lines the time needed to perform the same task (copying a certain file) has been measured, the difference between the executions is the size of the intermediate buffer used.

1.3.2. Run test lib.sh. Analyze the differences between the times shown by test.sh and those of test lib.sh. Can you justify them? Remember that benchmark.c and benchmark lib.c do the same task on the same source file.

2. Module 2. Memory [From March 25 to April 8] (45% = 20% + 25%)

2.1. Command top and executable call execution. Test the following questions. Attach screenshots showing the results of running top on your machine. What is the meaning of the columns VIRT, RES and %MEM?.

2.1.1. Find out how you can sort the information displayed by top and the value of each of these columns. Indicate which process has the largest value of VIRT and which of RES. It is necessary that it is always the same?.

2.1.2. We provide you the call.c program. Analyze its implementation, compile and run it. While it is running, analyze the information displayed by top regarding this process and how it's progressing. Can you explain how the information displayed by top evolves and why the OS decides to abort the process? If the OS takes too little time/too much time to abort the process, edit the source code and increase/decrease the value of the WAIT symbol and recompile the code.

2.2. Dynamic memory:

We provide you the fact.c program skeleton. This program will receive as a parameter an integer from the command line (with a maximum value of 20) and will write the factorial numbers from 0 to this number.

2.2.1. Study the source code provided. It's a skeleton program that you will have to complete. Notice that the memory space required to store the series number table should be created dynamically. Notice also that it requires to store the numbers in two formats: as a 64-bit integer and as a string (which should also be dynamically created based on the number of decimal dígits that the number has).

2.2.2. Complete the fact.c code so that it requests memory and calculates the factorial numbers table without wasting space by storing the factorials as strings. Once filled in, the supplied code will display it on the screen. Once displayed, the code will explicitly release all memory that has been requested. An example of the desired result is attached.

```
[enricm@willy dev]$  ./fact 20
0 1 1
1 1 1
2 2 2
3 6 6
4 24 24
5 120 120
6 720 720
7 5040 5040
8 40320 40320
9 362880 362880
10 3628800 3628800
11 39916800 39916800
12 479001600 479001600
13 6227020800 6227020800
14 87178291200 87178291200
15 1307674368000 1307674368000
16 20922789888000 20922789888000
17 355687428096000 355687428096000
18 6402373705728000 6402373705728000
19 121645100408832000 121645100408832000
20 2432902008176640000 2432902008176640000
[enricm@willy dev]$
```

Observations:

- Your code starts here */ i /* Your code ends here */. • Your code must be located between the fact.c lines /* Your code starts here */ and /* Your code ends here */.
- You can't modify the rest of the skeleton provided and must let the supplied code show the table contents.
- The sprintf routine can be useful to conver a integer number format to string format.
- The log10 routine can be useful to calculate the number of decimal digits in a number. To use it, you need to add -lm when compiling, in other words, gcc prog.c -lm -o prog.
- Before ending the execution of the program, it's necessary to explicitly release all the dynamic memory that has been requested.
- You will have to deliver the program source code and a screenshot that shows how it works.

3. Module 4: In/Out [From April 9 to 14] (15%)

The args.c program shows the parameters list that it receives from the command line. Several execution examples are attached:

```
[enricm@pcmartorell-i-ribas dev]$  ./args
argc = 1
  argv[0]=./args
[enricm@pcmartorell-i-ribas dev]$  ./args a1 a2
argc = 3
  argv[0]=./args
  argv[1]=a1
  argv[2]=a2
[enricm@pcmartorell-i-ribas dev]$  ./args a1 a2 > xxx
argc = 3
  argv[0]=./args
  argv[1]=a1
  argv[2]=a2
[enricm@pcmartorell-i-ribas dev]$  ./args a1 a2 2> xxx
[enricm@pcmartorell-i-ribas dev]$  ./args /bin/l??s*
argc = 6
  argv[0]=./args
  argv[1]=/bin/less
  argv[2]=/bin/lessecho
  argv[3]=/bin/lessfile
  argv[4]=/bin/lesskey
  argv[5]=/bin/lesspipe
[enricm@pcmartorell-i-ribas dev]$
```

Study the code, compile it, and verify that it works as indicated.

Answer the following questions justifiably.

3.1. Why does the third example argc have the value 3 and not 5?

3.2. Run the program args passing as a parameter the list of files in the /bin and /sbin directories where the name has exactly two characters.

3.3. Read the tee order in the manual. Use it to cause the result of the args command to be written into two files.

Resources

- Modules 1, 2, 3 and 4 of the subject.
- The classroom "Operating Systems Laboratory" (questions about Unix, C,...).
- "UNIX command interpreter" document (available in the classroom) or any other similar manual.
- Any basic C language manual.

Format and delivery date

It will be delivered in a **zip** file named with your campus ID and containing a pdf file with the answers to the questions and the program source code.

Delivery deadline: midnight on April 14, 2024