Fundamentals of Programming

Computadors – Grau en Ciència i Enginyeria de Dades – 2020-2021 Q2

Facultat d'Informàtica de Barcelona – Departament d'Arquitectura de Computadors

In this laboratory session, we will use several tools to analyze the execution of programs: gprof and gdb. We will learn how to use these tools in conjunction with a brief introduction to Git repositories and autotools toolset.

Getting experience with repositories

Download the support files from the S9 laboratory session (FilesS9.tar.gz), http://docencia.ac.upc.edu/FIB/GCED/COM/documents/Lab/FilesS9.tar.gz.
Create a folder named "ProjectS9" in the working directory that you usually work with in the COM Laboratory. Unpack the source code into this folder.

In this first part of the Lab you will get a brief experience of Git as a Version Control Tool. For more information, you can go to https://git-scm.com/docs

Check that you have the git command installed. If not, you can install it with (Ubuntu):

```
$ sudo apt-get install git
```

Exercise 1 – Create a repository

Type the following command line to create a repository held in the "ProjectS9" folder:

```
$ git  init
```

From now on, there is a hidden folder named ".git" (it is hidden because the first character is a ".") that holds all the required data for version management. You can add the tracking of the files that you select. To do this, you just add the files that you want to track, for example "fork.c" (you can add as many files as you want at once), as follows:

```
$ git  add  fork.c
```

Before sending the file to the repository (in this lab session it is held in your own computer, not in a remote server), you have to introduce few data about you to identify the person who performs the development:

```
$ git  config  --global  user.name  "MyName"
$ git  config  --global  user.email  "myname@upc.edu"
```

Finally, you can update the repository management system with the added files using a single commit:

```
$ git  commit  -m  "This is the first file"
```

Now, modify a single line of the file with a text editor, save the modified file, and perform again the steps "git add fork.c" and "git commit –m "This is a modification"".

You can check the repository modifications with "$ git log", whilst you can check complementary data with "$ git reflog".

A key element of Git Systems is the "HEAD" that indicates where are you. With the following steps you will see how easy can be to go to other versions of your code. Type the command line:

```
$ git  checkout  XXX
```

Where XXX is the alphanumeric label (first column of the "git reflog" output) that identifies the version you want to go to. Afterwards, go to the text editor and reload the "fork.c" file. You will see the original implementation you committed in that particular version. Now, you can type:

```
$ git  checkout  YYY
```

Where YYY stands for the alphanumeric label of the second version you committed. Afterwards, check again the text editor and reload the "fork.c" file and you will see that the code is again the one you modified and committed in the second version.

There are many things you can do with Git systems and this exercise pretends to be an introductory example. If you wish, you can keep adding the following source files (not the binaries) to the repository to get additional experience. It is totally voluntary.

Before continue, compile the codes with "$ make".

Exercise 2 – understanding your source code – fork.c

Look at the source code of "fork.c", understand it.

Run the program and explain what the program does.

Write your answer in your new *answers.txt* file for this session.

Analyzing the execution with gprof

gprof is already installed in our machines:

```
$ gprof
a.out: No such file or directory
   # it means that by default it uses the "a.out" binary file for instrumentation analysis
$ man gprof      # get information on the use of gprof
```

Programs need to be compiled with the instrumentation "-pg" option (profiling with gprof). Change the Makefile to compile the program with the -pg option.

"-pg" indicates to the compiler that it should generate additional code to collect and write information suitable for gprof during execution. The information will be written to the *gmon.out* file.

Execute the program and analyze the information you can obtain. You can use the Makefile rule gprof-fork:

```
$ make gprof-fork
```

will obtain the output corresponding to

-Flat profile

-Call graph

-Function reordering

Exercise 3 – flat profile

Analyze the information provide by the first gprof command:

```
$ gprof -b  -p   fork   gmon.out
```

You can read also some details about the output information provided by not using the *brief* (-b) option:

```
$ gprof    -p   fork    gmon.out
```

Does the number of calls reported by gprof agree with the total number of calls made by the program to the different functions? Why?

(hint: GMON_OUT_PREFIX=prefix-string environment variable, use “export” (bash) or “setenv” (tcsh))

Exercise 4 – call graph

Analyze the information provide by the second gprof command:

```
$ gprof -b  -q   fork   gmon.out
```

Also look at the information provided with the command:

```
$ gprof    -q   fork   gmon.out
```

Describe the call graph obtained.

Exercise 5 – function reordering

Analyze the information provided by the third gprof command:

```
$ gprof    -r   fork   gmon.out
```

Describe what is the use of the function reordering. (Hint: man gprof, see --function-ordering option description).

The “sort” program

Look at the mysort.cpp program. Understand it, make sure it is compiled and execute it:

```
$ make mysort

$ make

$ ./mysort 100000
```

Exercise 6 – flat profile

Get the flat profile of the previous execution of mysort, and list the functions reported by the flat profile and their execution time. You can use the “--demangle” option of gprof to get inteligible function names.

Exercise 7 – flat profile with no optimization

Compile the mysort program with no optimizations (-O0), and execute it to obtain the flat profile. You may need to reduce the amount of numbers to be sorted due to the low performance achieved by the non-optimized version (try 32768). Explain why the number of functions listed now is so different.

Debugging programs with GDB

GDB is the GNU Debugger, the tool to find bugs in your programs. With gdb, programs can be executed on a controlled environment, where the debugger can start and stop the execution of the processes, examine and modify symbols and memory locations, look at the source code and machine instructions, among many other options.

Exercise 8 – debugging and accessing symbols (fork)

Analyze the information obtained when executing "fork" under tracing with:

```
$ gdb fork
```

First, we will add a breakpoint on the call to the function "puts". This way, every time the process invokes "puts", the executions will be stopped, and GDB will give you the control of the session. Do:

```
(gdb) break puts
```

Write the information that GDB displays regarding the new breakpoint in your "answers.txt". What does it mean the message? (answer it also in answers.txt).

Now you can run your program with the "run" command:

```
(gdb) run
```

The process is started and it will stop in the breakpoint. Write and explain in your "answers.txt" file the information displayed by GDB upon stopping the program. Observe that part of this information consists of the file and line number where the process is executing. Does GDB find the file? Why?

Now, let's display more information about the execution: where we are?

```
(gdb) where
```

Write the information you obtain in the "answers.txt" file. How many frames are reported by GDB? What is the meaning of each frame? Where are the frames stored in your process?

Before continuing with the execution, observe that the program in fork.c defines the "i" variable to take the initial value of 99. Can you determine how to check that the "i" variable has been properly initialized? (Hint: "up" command). Write your findings in your "answers.txt" file.

Practice with the "up", "down", "list", and "print" commands. See the listing of the frame #0 (the main program), and see the listing of the #1 function (__GI__IO_puts, probably not available in our systems). Print the values of the variables in the main function: "pid", "i", "res". What happens to "res" and "pid"?

Print the contents of the "message" variable. What do you see? Why? (Hint: is it initialized?).

Print "N", and the contents of "vector". What happens with "vector". Can you print individual elements from "vector"? (Hint: GDB understands C/C++ expressions).

Exercise 9 – running with the debugger (fork)

Now you are in the position to keep executing the process through the 3 iterations it does. The goal is that everytime it stops in a "puts" message we can look at the value of the "i" variable and see that its value evolves from 0 to 2:

```
(gdb) cont      # for continue, keep running till reaching the next breakpoint
```

As there are 2 "puts" invocations per iteration in the process, the execution will stop twice per iteration, and we should be able to see the values of "i": 0, 0, 1, 1, 2, 2… end of the process. Write in your "answers.txt" file, the commands you use to go from iteration 1 to iteration 2, while showing the variable "i" (Hint: be careful because the child process created by the program will print a message after GDB displays its prompt (gdb), and if you type enter – an empty line -, GDB will automatically repeat the previous command – cont, advancing till the next breakpoint).

Exercise 10 – improving the runs with the debugger (fork)

Now, we will learn to stop at particular lines, by inserting a breakpoint at a line in the main function, and automatically display variables upon stopping. Start a new debugging session:

```
$ gdb fork

(gdb)
```

Use list till reaching the lines executed by the parent process (pid != 0):

```
40      else if (pid > 0) {
41        sprintf(message, "parent: Hello, World");
42        puts(message);
43        add_to_vector(vector, N, 5.0);          ← Insert a breakpoint before calling add_to_vector
44        res = waitpid(pid, NULL, 0);
```

How do you insert a breakpoint before the execution of the call to "add_to_vector"? Explain it in "answers.txt" file. (Hint: while listing the program, use the line number in the "break" command)

Can we display the value of symbols automatically upon stopping? (Hint: "display" command). Explain it in your "answers.txt" file.

Prepare this execution environment, so that you can repeat the execution with 1 "run" (starting the process) and 3 "cont" to reach the end of the execution, and check that the variable "i" is displayed automatically, taking the values 0, 1, and 2.

Exercise 11 – check vector data (mysort)

Now, let's practice with the debugger and mysort. Start a GDB session for mysort, and list the program. Let's insert two breakpoints:

- In functions "mysort", and "mymerge"
- In the line "correct = check_sorted(v);" (number 150 in our example)

Run the program, for example with 20 numbers, and wait till it stops on the "mysort" breakpoint. Look at the vector "v", and determine its contents. Are they sorted? Write your findings in your "answers.txt" file.

Observe that the last parameter of "mysort" (h) receives the value 20 in this first invocation. Set the vector "v" to be displayed at each stop for both "mysort" and "mymerge", and with "continue" observe how it is being sorted at each recursive call and return. Write your comments about how the vector is sorted in your "answers.txt" file.

Upload the Deliverable

When done with the lab session, to save the changes you can use the tar command as follows:

```
#tar  czvf  session9.tar.gz   answers.txt  Makefile*  *.c  *.cpp *.h *.out
```

Now go to RACO and upload this recently created file to the corresponding session slot.