Fundamentals of Programming

Computadors – Grau en Ciència i Enginyeria de Dades – 2019-2020 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 uftrace. 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).
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

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.

Analyzing the execution with uftrace

See Appendix A in order to do the installation of uftrace.

uftrace also needs to have the application compiled with "–pg" in order to obtain more accurate information. As we have compiled "fork" already with "-pg", we can reuse the same binary.

uftrace is able to work in several modes, allowing different types of experiments and information to be obtained:

- Live: trace functions of a program during a live execution
- Record: run a command and record its trace data
- Info: obtain information from the recorded trace data
- Report: obtain statistics from the recorded trace data
- Graph: display the call graph from the recorded trace data
- Replay: show a previously recorded trace data
- Dump: see the raw data information from the recorded trace
- Recv: receive information from another process and save it as a trace
- Tui: text-based user interface to navigate the graphs
- Script: associate python2.7 commands to program events

Exercise 8 – live tracing (fork)

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

```
$ uftrace live –column-view  ./fork
```

Explain in the *answers.txt* file how the uftrace tool allows to distinguish among the different processes.

Exercise 9 – live tracing (mysort)

Analyze the information obtained when executing "mysort" when tracing it with:

```
$ uftrace live ./mysort 300
```

(hint: search for the "mysort" and "mymerge" functions in the output of uftrace).

Explain in the *answers.txt* file how the uftrace tools allows to determine that a function call is recursive.

Exercise 10 – record tracing (mysort)

Record a trace of mysort:

```
$ uftrace record ./mysort  1024
```

Explain in the *answers.txt* file the output of the uftrace tool, when using the "report", "graph" and "tui" commands.

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  *.h
```

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

Appendix A: Installing uftrace

We will install *uftrace* from source code (https://github.com/namhyung/uftrace). First, download it from:

https://packages.debian.org/sid/utils/uftrace

-> http://deb.debian.org/debian/pool/main/u/uftrace/uftrace_0.9.0.orig.tar.gz

On a directory separated from FilesS9, uncompress the package:

```
$ gunzip <uftrace_0.9.0.orig.tar.gz | tar xf -
```

In case you have major problems with the following procedure, you can install uftrace with the following command line:

```
$ sudo  apt-get  install  uftrace
```

NOTE: *you will have to introduce the password of the user itself. For example, in the Ubuntu VM image from the FIB, the password should be "sistemes".*

PROCEDURE to compile and install:

Configure the tool:

```
$ cd uftrace-0.9/
$ ./configure ... # see file hints.txt, and adapt the "configure" command appropriately
                  # to your bash or tcsh environment,
                  # so that it installs uftrace in the directory $HOME/dades/linux/uftrace-install
```

Check that all options are active ("on") in the report obtained after configure.

Compile:

```
$ make
```

Install:

```
$ make install
```

Set your PATH environment variable to point to the installation directory binaries:

```
$ export PATH=$HOME/dades/linux/uftrace-install/bin:$PATH     # (bash)
$ setenv PATH  $HOME/dades/linux/uftrace-install/bin:$PATH     # (tcsh)
```

Set also your MANPATH environment variable to point to the manual pages installed with uftrace:

```
(bash)  $ export MANPATH=$HOME/dades/linux/uftrace-install/share/man:$MANPATH
(tcsh)   $ setenv MANPATH  $HOME/dades/linux/uftrace-install/share/man:$MANPATH
```

You can also add these commands to the ~/.bash_profile and ~/.tcshrc to make them permanent across working sessions.