Libraries and Compilation Environment (II)

Computadors – Grau en Ciència i Enginyeria de Dades – 2021-2022 Q2

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

This session complements the previous Lab session focused on Libraries and Compilation. In particular we will pay special attention to deal and compare a compiled program (based on C) and an interpreted app (based on Python). The source code is the same Fibonacci algorithm include in the previous session. We will get experience with debugging and comparing the cost of executing both codes.

Write your answers to the following exercises on a text file, called **"answers.txt"**, as usual.

The sample programs

In this session we will use two similar versions of the Fibonacci program, written in Python and C, used in the previous Lab Session. Download the program set from:
https://docencia.ac.upc.edu/FIB/GCED/COM/documents/Lab/S7/FilesS7.tar.gz

Take some time to view and understand both codes before start working with them.

Debugging Python programs

Execute the Python version of Fibonacci with the Python debugger:

```
$ python3 -m pdb  ./fib.py  4
```

Invoking Python with the "-m pdb" option automatically invokes the python debugger that is loaded from the "pdb" module. Be sure to provide "4" as the number argument for fib.py, as indicated.

The fibonacci program is automatically started, and the debugger shows a little bit of context (file and line number, and next line to execute), and waits for commands:

```
> /home/alumne/Documents/COM_GCED/Lab/S7/fib.py(2)<module>()
-> import sys
(Pdb)
```

(Pdb) is the python debugger prompt. Try help to see the debugger commands.

```
(Pdb) help
```

Use help to list the help for some of the useful commands: list, next, step, where, up, down, display... for example:

```
(Pdb) help next
 ...
(Pdb) help step
```

Observe that "next" executes till the next sentence, jumping across function calls, while "step" will enter inside function calls.

Exercise 1

Usually it is interesting to know where in the program we are executing with the debugger.

Which command would you use to "list" the program source?

How do you know the exact line where the execution has stopped?

Apparently, with the python debugger you can only list lines from the current file. Commands "up" and "down" change the stack frame to be examined (active), and also the file listed, which is always the one associated to the active stack frame.

Observe that in the case of the python debugger, the execution automatically stops at the beginning. This happens because the python debugger manages breakpoints automatically. Additionally you can add breakpoints with the "break" command of the debugger.

Right after starting the execution (`python3 -m pdb  ./fib.py  4`)
Issue a pair of "next" commands, till the debugger displays the sentence:

```
-> argc = len(sys.argv)
```

Exercise 2

Now look into the values of the variables "sys.argv", and "argc":

Which command do you use to "display" or "print" their values?
What is the value of "sys.argv"?
What happens with the value of "argc" if the line shown has not be executed yet?

Observe that you can examine the values of the symbols in the current stack frame, and in the case of Python, only when they have been actually defined. Again, the commands "up" and "down" will allow you to navigate across the stack frames and examine the symbols (local variables and arguments) defined on them.

Observe also that you can use "display ..." and "print (...)" to examine the value of symbols. "display" is a debugger command that keeps a list of symbols to be displayed when they change their value. "print" is the python "print" primitive. For this reason in Python3, "print (...)" requires the use of the parenthesis, to conform to the Python3 syntax.

Exercise 3

Now issue another "next", and you should be able to see the value of "argc":

What is the new value of "argc"?
Has it been displayed automatically? Why?

Now continue the execution with the "step" command. This will make the debugger to enter inside function calls:

```
 (Pdb) step
> /home/alumne/Documents/COM_GCED/Lab/S7/fib.py(16)<module>()
-> num = int(sys.argv[1])
```

Issue "step" commands until the program enters the "fib" function:

```
 (Pdb) step
--Call--
> /home/alumne/Documents/COM_GCED/Lab/S7/fibonacci.py(2)fib()
-> def fib(n) :
 (Pdb)
```

Exercise 4

Now, try to display the variable "num", that was assigned in the main function a few sentences ago:

Why can't you see the value of "num"?

What can you do to access that variables? (hint: commands up and down)

Explain how you access the value of "num", and then continue with the "step" commands.

Exercise 5

Now issue 6 consecutive "step" commands, and determine at which level of the recursion the program is.

Which command do you use to determine the recursion level? (hint: where)

How many levels "up" do you need to go to reach the main function, and be able to display the variable "num" again?

Go "up" the number of levels needed to reach the main function, and display the variable "num".

Now execute alternatively the "step" and "display n" commands, in order to see how the "n" parameter of fib(...) evolves. At the end of the debugging session with the python debugger, you can issue the "continue" command to reach the end of the program and see the result:

```
(Pdb) cont
fibonacci( 4 ) = 3
The program finished and will be restarted
```

Observe that the program is restarted automatically within the python debugger. Use "quit" in order to end the debugging session.

Exercise 6

Now restart the execution of the python program, and advance till this sentence (if (argc != 2)):

```
(Pdb) next
> /home/alumne/Documents/COM_GCED/Lab/S7/fib.py(9)<module>()
-> if (argc != 2) :
```

*At this point, change the value of argc to 1 (argc=1), and continue with the execution. Explain your findings in your **"answers.txt"** file.*

You can find more information about the python debugger on the Python documentation:

https://docs.python.org/3/library/pdb.html

Debugging C programs

Now look into the execution of the fibonacci program in C with the GDB debugger. Start the debugging session:

```
$ gdb ./fib
```

and:

```
(gdb) run 6
```

Observe that with GDB, the parameters of the program are given with the "run" command. And also observe that, unlike in the Python case, the program...

- is not automatically started until the user issues the "run" command with its arguments.
- is run till the end, as no breakpoints are set implicitly by the debugger.
- is stopped only in case that a problem occurs.

while inside gdb, re-issue "run 6" to execute the application with 6 as a parameter. Fibonacci should run freely till the end and show the result (8).

You can also list the source code. As this version of fibonacci consists of two files (fib.c, with the main function, and fibonacci.c, with the fibonacci function), listing functions will switch automatically the file on display.

```
    (gdb) list main
    ...
    (gdb) list fibonacci
    ...
```

In case while listing a function, GDB does not show the first lines of it, you can use

```
    (gdb) list -
```

to see the previous lines.

In order to interrupt the execution of Fibonacci at specific points, we can:

- set a breakpoint in a C source line in the current file:

  ```
  (gdb)  break <line-number>
  ```

- set a breakpoint in a C source line on a specific file:

  ```
  (gdb)  break <filename>:<line-number>
  ```

- set a breakpoint at a function entry-point

  ```
  # replace <function-name> with the name of the function
  # in your program, where you want the execution to stop
  (gdb)  break <function-name>
  ```

When debugging a C program, we usually set a breakpoint in the “main” function, so it stops at the beginning of the user code:

```
      (gdb)  break main
```

For the next exercises, you should:

- set a breakpoint in the source line (find which is its line number using “list”):

  ```
  num = strtol(argv[1], NULL, 0);    (fib.c)
  ```

- set another breakpoint in the entry-point of the function "fibonacci" (fibonacci.c).

You will use some of these additional GDB commands:

- **cont**, to continue with the execution till the next breakpoint, or the end of the program
- **next**, to execute to the next program sentence without stopping inside function calls
- **step**, to execute to the next program sentence, stopping inside function calls
- **finish**, to execute till the current function returns, or the next breakpoint

Exercise 7

- *How do you set the breakpoint in the line*
  `num = strtol(argv[1], NULL, 0);    (fib.c)   ?`
- *How do you set the breakpoint in the function "fibonacci"?*

Exercise 8

Now you can run the program with "run 6", and when stopped in the line

```
num = strtol(argv[1], NULL, 0);    (fib.c)
```

execute it (next), and check that the resulting value (num) is actually 6.

*Write down the commands you use to do this in your **"answers.txt"** file.*

Exercise 9

Now you can execute till the next breakpoint (this time the execution will be entering the fibonacci function). And then use finish as may times as needed to reach the return of the last call to fibonacci. Given a fibonacci of 6, how many "finish" invocations do you need to issue?

You can check with these results we got:

fib(2) -> 3 "finish"	fib(3) -> 5 "finish"
fib(4) -> 9 "finish"	fib(5) -> 15 "finish"
fib(6) -> 25 "finish"	fib(7) -> 41 "finish"
fib(8) -> 67 "finish"	

This shows that the recursion is everytime depper, and depper...

Exercise 10

Now, without leaving from the GDB, re-execute the fibonacci sample, with parameter 10:

```
(gdb) run 10
```

Be sure that the breakpoint at entry of function "fibonacci" is still there. You can use the command:

```
(gdb) info break
2       breakpoint     keep y   0x00005555555551d0 in fibonacci
```

Then use "continue" to reach 6 levels of depth into the recursion

```
(gdb) cont
...
Breakpoint 2, fibonacci (n=n@entry=4) at fibonacci.c:5
```

Now display the value of the parameter "n" at the various stack frames:

- *How do you display the value of the parameter "n"?*
- *How do you go "up" to the next stack frames?*
- *What are the values of the argument "n" in the various stack frames?*

Afterwards, "delete" the breakpoint, and let the program run to the end

- *How do you delete the breakpoint?*
- *How do you let the program "continue" to the end?*
- *Is the result still correct?*

*Write your findings in your **"answers.txt"** file.*

Exercise 11

What is the difference between the commands "print" and "display"?

(Hint: look at the "help print" and "help display" in gdb)

Exercise 12

Now start another execution of the fibonacci program within the GDB debugger, and make it stop at the line:

```
        if (argc != 2) {
```

Change the value of the "argc" variable (for example, set it to 1), and continue with the execution of the program...

- *How do you change the value of the "argc" variable ? (Hint: see the "set" command)*
- *Explain in your "answers.txt" file what is the consequence on the execution of the program when you "continue" it, of having modified the "argc" variable.*

You can find more information about GDB on the GNU documentation:
https://web.mit.edu/gnu/doc/html/gdb_9.html

Comparing the cost of executing compiled versus interpreted code

Deal with the following exercises to better understand the difference between running interpreted and compiled codes.

Exercise 13

In order to compare the different optimizations performed in C-compiled codes, modify the ***Makefile*** *to include three different rules to compile the Fibonacci code using the optimizations "-O0", "-O1", "-O2", and "-O3". Name the resulting binaries "fib0", "fib1", "fib2", and "fib3", respectively. It is not necessary to keep the "-g" flag. (HINT: check one of the Makefiles of the previous session).*

The following exercises are done to calculate "Fibonacci 30". If your computer is high performance, you can increase the number to be computed.

Exercise 14

Use the command "/usr/bin/time" (it is used as "/usr/bin/time COMMAND") to compare the execution time of "./fib.py 30" and "./fib 30". In the case of the C-compiled code, do it with the four diferent binaries. Write the comparison of execution time (elapsed time) among the executions.

Exercise 15

Use the valgrind command as follows to measure the number of instructions executed ("I refs: number"):

```
#>valgrind  --tool=callgrind  COMMAND
```

Let's repeat the same execution than the previous exercise (fib 30) with the python and the four diferent optimized C-compiled codes. Write the comparison of number of Instructions executed among the executions.

Complementary note: those of you that want to do complementary analyses, you can execute:

```
#>kcachegrind  cachegrind.out.xxx
```

where "cachegrind.out.xxx" are the files generated by the valgrind executions of exercise 15. This tool shows you a detailed and visual analysis of the measurements.

Upload the Deliverable

To collect all contributions to this deliverable, you can use the tar command as follows:

```
# tar  czvf   session7.tar.gz    answers.txt   Makefile
```

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