operating systems lab - week 7: exercise

Nicolo Colombo

November 9, 2020

This week, no new concepts have been introduced. Video 7 is a tutorial on how to write a program that mimics the behaviour of the UNIX shell. The program, `intCalc.c`, can be seen as a concrete implementation of some of the concepts you have seen in the past weeks, in particular Week 3, Week 4, and Week 5.

During this lab session, you can watch the live-coding videos mentioned in Slides 7 and stop the videos to ask questions if some parts are not completely clear to you.

You can also copy, study, and run the step-by-step pieces of code provided in Slides 7 or `intCalc.c`.

Finally, you can work on previous weeks' exercises, if you have not completed all sections. In particular we suggest you focus on Exercise lab-sheet 3 and Exercise lab-sheet 5, as these contain the most challenging questions. Please try to watch all videos and understand, compile, run and possibly rewrite all provided codes before attempting this week's Moodle quiz

Lab Quiz 7

as some of the quiz questions may require you to run, comment or modify the programs you are asked to write.

Set up

Depending on your OS, use the following instructions to connect to `linux.cim.rhuk.ac.uk`:

Unix Open the terminal and run

```
ssh yyyyxxx@linux.cim.rhul.ac.uk
```

where `yyyyxxx` is your college username, and enter your password to access the teaching server.

Windows Launch the Windows SSH client `puTTY` [1], enter the following

```
linux.cim.rhul.ac.uk
```

in the empty field *Host Name (or IP address)* and click on *Open*. The client opens a new window where you are required to enter your college user name `yyyyxxx` and password.

Once logged in, you should be able to see the content of and navigate in your home directory using the standard UNIX commands, e.g. `ls`, `cd`, `cp`. Go to the `CS2850labs` directory[2] and run the command

```
$mkdir week7
```

to create a new sub-directory called `week7`. We suggest you save and compile all programs you write for this exercise in this directory.

Use a command-line text editor, e.g. emacs, nano, or vim,

to open, edit and save your programs. The advantage of command-line editors is that they can be used in a non-graphical SSH session. [3] You can create a new C file or open an existing C file, `file_name.c`, by running the command

```
$editorName file_name.c
```

[1] `puTTY` should be installed on all department's machines. If you work on your own Windows machine you can download it at download puTTY and install it as explained.

[2] The parent directory you have created for the first week's lab

[3] If you do not want to open and close the editor every time you modify and save your code you can open a new SSH session and use two shell windows simultaneously.

where, e.g. `editorName` is `vim`

We suggest you save separate files for all single parts of this exercise and follow the name suggestions given in each section. [4]

Compile your C code by running

```
$clang -o file_name file_name.c
```

and run the corresponding binary files `file_name` through

```
$./file_name
```

For debugging, we suggest you use the free debugging tools of valgrind, which is already installed on the teaching server `linux.cim.rhul.ac.uk`. To check your code, you just need to run the command

```
$valgrind ./file_name
```

and have a look at the messages printed on the terminal.

1 Watch the live-coding videos

An *interactive program* is a program that requires user interaction to operate. The classic example of an interactive program is the UNIX shell. In Week 5, you have seen in some details how the UNIX shell works and the structure of the underlying program. The integer calculator implemented in `intCalc.c` is another example of an interactive program.

The general structure of `intCalc.c` is very similar to the general structure of the shell. As for the shell, `intCalc.c` allows the user to interact with the program during the execution. Obtaining such interactive behavior can be challenging from an implementation perspective, as you need to write many different subroutines and organize a pretty long series of instructions.

A good approach when you implement a highly-structured program from scratch is to proceed step-by-step by splitting the target task into simpler elements. In this case, we require the final program to:

1. be interactive (`step1.c`)
2. read from the terminal (`step2.c`)
3. clean the user's input (`step3.c`)
4. transform the input into integers (`step4.c`)
5. perform selected arithmetic operations (`step5.c`)

Each task has been implemented in `step#.c` and is described in Slides 7. The following live-coding sessions show you how to write each `step#.c` from scratch:

1. video 1: live coding of `step1.c` [7:08]
2. video 2: live coding of `step2.c` [8:52]
3. video 3: live coding of `step3.c` [6:40]
4. video 4: live coding of `step4.c` [19:56]
5. video 5: live coding of `step5.c` [13:23]

We suggest you try to implement your version while following the video and stop if something is not completely clear to you. Try also to add, remove, or slightly change some of the program lines and see what happens when you compile or execute the code. This is the best way to understand the role of each instruction, spot errors, or find better solutions.

Finally, note that the given code is only one of many (and possibly better) implementations of an interactive calculator. Feel free to ask questions about anything that you may find odd or unnecessary, and propose alternative solutions.

[4] This is mainly because some of the Moodle quiz questions may refer to single pieces of code through the suggested file names.

2 Work on Week 3 and Week 6 exercise (optional)

As you may have noticed, some of the functions used in `intCalc.c` are similar to those you wrote for the Week 3 and Week 6 lab-sheets. For example,

- in Section 1 of Week 3 exercise, you were asked to write functions similar to `readLine`, `getIntegers` [5], and `getInt` defined in `intCalc.c`.
- in Section 2 of the same exercise, you were asked to write `getWords`, which is the analogous of `getIntegers` for strings. [6]
- in Section 1 of Week 6 exercise, you can now use the improved version of `readLine` defined in `intCalc.c` and, if your are not completely happy with your old implementation of `getWords`, you can use the structure of `getIntegers` to write a new version of it.

3 Improve `intCalc.c` (optional)

If you are confident about your understanding of the code given in `intCalc.c` you can try one or more of the following (in the order):

- to rewrite `intCalc.c` from scratch without looking at the provided code
- to include the *modulo* operator, `%`, in the set of the possible operations
- to set a *default operation* and, if the user changes it to something else, switch back to that after one iteration with the new one
- to allow the program operate with more than two integers e.g.

  ```
  12 34 56
  ```

  may produce

  ```
  12 + 34 + 56 = 102
  ```

[5] In Week 3 we call it `loadVector`

[6] Note that an uncompleted version of `getWords` was given in Question 4 of Lab Quiz 3. Now, you may find easier to answer that question by comparing the question's code with `getIntegers`