FIT2100 Assignment #1 Building a Graphical Shell with C Programming Semester 2 2018

Mr Daniel Kos
Admin Tutor, Faculty of IT.
Email: Daniel.Kos@monash.edu

August 1, 2018

Revision Status

$Id: FIT2100-Assignment-01.tex, Version 1.0 2017/08/12 18:00 Jojo $

$Id: FIT2100-Assignment-01.tex, Version 2.0 2018/06/18 18:00 Daniel $

$Id: FIT2100-Assignment-01.tex, Version 2.4 2018/07/25 18:30 Daniel $

$Id: FIT2100-Assignment-01.tex, Version 2.5 2018/07/31 11:45 Jojo $

Based on command-line interface assignment prepared for FIT2100
in 2016-2017 by Dr Jojo Wong.
Adapted to graphical shell assignment by Daniel Kos.

Contents

1 Introduction

This assignment is due on **24th August 2018 (Friday) by 5:00pm**. It is worth **15% of the total unit marks**. A penalty of 5% per day will apply for late submission. Refer to the FIT2100 Unit Guide for the policy on extensions or special considerations.

Note that this is **an individual assignment** and **must be your own work**. Please pay attention to Section 3.2 of this document on the university policies for the *Academic Integrity, Plagiarism and Collusion*.

This first assignment consists of **three main tasks** and all three tasks should be submitted as **three separate C programs** with supporting documentation on usage. All the program files and any supporting documents should be compressed into one single `.tar.gz` file for submission. (The submission details are given in Section 3.)

Assessment: For the working program and code architecture components of your assignment 1 mark, the 3 tasks carry unequal weightage of marks with 35%, 40% **and** 25% **respectively.**

2 Graphical Shell

2.1 Background

You are currently getting familiar with the `bash` command line interface. Bash is an example of a *shell utility*. A shell is an important component in a typical computing environment, which allows the user to interact with various functions of the operating system.

Bash works by waiting for user input (in a loop), and responding to each command the user types into a TTY terminal by doing the action the user requests (then continuing to the next iteration of the loop, so the user can then input another command, repeating until the user decides to quit). However, learning to use a command-line interface can be frightening to people who aren't good with computers. Many operating systems such as Windows and macOS provide the user with a *graphical shell* so novice users can interact with files using a mouse. (Examples are Windows File Explorer and also the Finder in macOS.)

2.1.1 Objective

The objective of this assignment is to implement a simplified version of a graphical shell like you might find in a popular operating system. This works on the same principles as a command-line shell, but allows the user to provide input in a graphical way by clicking file icons on the screen, browsing folders, etc.

Your shell shall be implemented on the Linux operating system in the C programming language, making use of external Linux utilities where specified, and producing output within the ioL graphical console environment.

2.1.2 Preparation

Throughout your degree up to now, whenever you have printed output from a program, the output was always displayed in a plain-text terminal window (technically known as a TTY). A standard TTY terminal is not ideal for building a lavish application. The output you print can only be displayed sequentially, line by line in plain text, and you can't display graphics or ask the user to click a button.

For Tasks 2 and 3, we will be replacing the traditional TTY with the ioL console system. You will compile your programs in the same way, but will run them inside a different environment.

Consider the line of C code below:

```
1 printf("Hello <span italic=true {World!}>\n");
```

If you run the program in the usual way (i.e. output is printed to a TTY terminal), the user will see everything you've put in the quotation marks. But if you run your program inside an ioL console, the user will see the words 'Hello *World*' with the second word displayed in *italic*. This approach provides the ability to make your programs more intuitive for the user. (You will need to put: '`iol -- `' in front of the command used to run your program, in order to make it run in the special environment, and the ioL console window will close automatically as soon as your program quits.)

To produce the graphical user interface in Tasks 2 and 3, you will need to print your program output according to the *ioL markup syntax*, including giving the user some way to provide input back to your program. ioL has special rules for how you need to print your output in order to make things show up the way you want. Your Linux virtual machine environment is already set up with the ioL console system needed to run these tasks.

A getting-started tutorial is available online at:

`http://doc.iol.science/getting-started/tutorial`

You should go through the tutorial before you attempt Tasks 2 and 3 of this assignment.

2.2 Task 1: Building the Essential Program Structure

In the first task, you will implement a very rudimentary command-line interpreter running in a TTY terminal, taking command input from the terminal as typed by the user and printing output back to the terminal. (This is not a graphical shell yet, but once you have the essential program structure in place, you will turn it into a graphical shell in Task 2.)

For this task, you should name your main source file as `task1shell12345678.c`, where 12345678 is your Monash Student ID number.

Implement your terminal shell, `task1shell`, as follows:

1. When the user starts your shell, it should print the filenames of all the files in the current directory. Do not print hidden files. For each filename, also print whether it is a directory, executable program, or ordinary file.
2. Your shell should then wait for the user to input a command from the following table and behave accordingly.
3. After processing each command, the shell should allow the user to enter another command, and repeat until the user decides to quit.

For this task, your program should implement the following commands:

Command	Description
`d:<directory>`	Change the current working directory to `<directory>`, and re-print the files at the new directory location. This command should also update the `PWD` *environment variable*. It should also respond correctly if `..` is specified as `<directory>`, by navigating to the parent of the current directory. If the user does not have permission to enter a directory, print an appropriate error message.
`x:<filename>`	Execute the specified `<filename>` by running the program in a new terminal window. Your program should run an instance of the `xterm` utility to create the new TTY terminal window. Your shell should not wait for the new process to finish, but should allow the user to continue entering commands while the new child process is running. If the file cannot be executed because the user does not have permission to do so, print an appropriate error message and do not open a new terminal window.
`v:<filename>`	Display the contents of the text file `<filename>` in a new `xterm` terminal window, making use of the `less` utility. If the file size is larger than 500kb, do not allow the user to open the file; instead print an appropriate error message to inform the user that the file is too large to be viewed in this way. Your shell should not wait for the new process to finish, but should allow the user to continue entering commands while the file is being viewed in the other window.
`q:`	Quit the shell.

Each command should print a reasonable error message if the operation could not be completed, without terminating your program.

Error handling: You are required to validate against the possible error conditions mentioned above. You are not required to validate that the user has entered a valid command in this part, since the graphical user interface that you will build in Task 2 will shield your program

from invalid inputs.

Robustness: Keep in mind that directory paths might contain spaces, and might be longer than you expect if the user wants to navigate to a path several folders deep. For the purposes of this assignment, you may assume that the number of characters in the user's paths/filenames will be fewer than `PATH_MAX` (this is a pre-defined constant in your C environment, but you will need to include `<limits.h>` in order to use it).

```
me@mycomputer:/home/me/FIT2100/$ ./task1shell
Current directory: /home/me/FIT2100/
***Directory listing follows***
File:       FIT2100 is awesome.txt
File:       MFW I get an HD.jpg
Executable: task1shell
File:       task1shell.c

d:..
Current directory: /home/me
***Directory listing follows***
Executable: a.out
Directory:  Documents
Directory:  FIT2100
File:       helloworld.c
Directory:  Pictures
File:       Reasons this unit is great.txt
Directory:  Videos

v:Reasons this unit is great.txt
Error: Could not open text file. File size too large.

q:
me@mycomputer:/home/me/FIT2100/$ _
```

Figure 1: Sample output for Task 1, when the user runs the shell from within `bash`. The user's input is shown in **bold**.

2.3 Task 2: Graphical User Interface

By modifying your Task 1 shell, implement a new version, `task2shell`, to run within an ioL console instead of a TTY terminal.[1]

For this task, you should name your main source file as `task2shell12345678.c`, where 12345678 is your Monash Student ID number.

Your Task 1 implementation produces output intended for display on a plain-text TTY terminal. Modify the print statements in your implementation to mark-up your output in a graphical way as follows:

- Instead of printing the directory contents in plain text, use visual icons to indicate the file types.[2] As in Task 1, the three different file types are directories, executable files, and ordinary files. The display should be updated whenever the user changes the current working directory.
- Add `onclick=` handlers[3] to your program's output so that when the user clicks on the file icons, the appropriate command for that file is carried out by the shell. For example, when the user clicks on a directory named `test`, the console should input `d:test` back to your shell program, and your program should print new output to display the new directory contents on the screen. Similarly, when the user clicks on an executable file, a similar `x:...` instruction should be produced, and so on. All other files should be treated as text files.
- Error messages should be displayed on the screen in an appropriate way (for example, you might print a `<box> element` at the top of the window for error messages).
- The user should have a way to navigate to the parent folder.
- The user should be able to quit the program gracefully by closing the window.

[1] The user should be able to run your program by typing `iol ./task2shell` or something similar in order to get the correct graphical functionality.

[2] You can either use your own icon images, or make use of special font symbols defined in the Unicode character set for this task. For example, the character code (in hexadecimal) for an open folder symbol is `x1F4C2`, and can be printed as `"</#x1F4C2>"`. A good resource for finding character codes for various symbols is `https://www.fileformat.info/info/unicode/block/miscellaneous_symbols_and_pictographs/list.htm`

[3] `http://doc.iol.science/fields/reference/onclick`

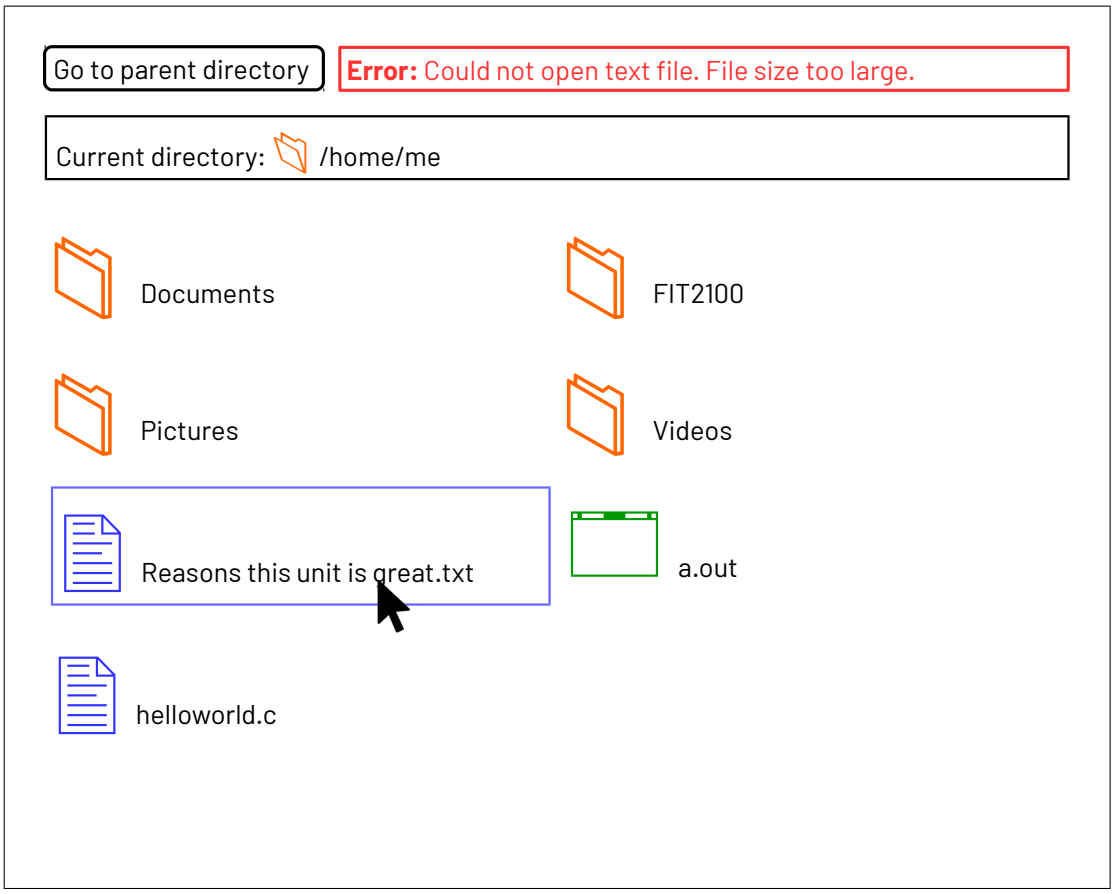

Figure 2: Sample output for Task 2, running through an `iol` console instance. This is just one idea — you may use your imagination to lay out your shell interface in any reasonable way you wish as long as you implement the requirements. The icons in this example are just special character symbols marked up in a large font size and different font colors.

2.4 Task 3: Advanced File Operations

In this task, you will add extra features to allow the user to carry out more advanced operations on each file. For this task, you may choose to make use of the `oncontextclick=` handler[4] in your marked-up output so that the user can right-click a file to display buttons for additional commands.

For this task, you should name your main source file for your advanced implementation as `task3shell12345678.c`, where `12345678` is your Monash Student ID number.

[4] `http://doc.iol.science/fields/reference/oncontextclick`

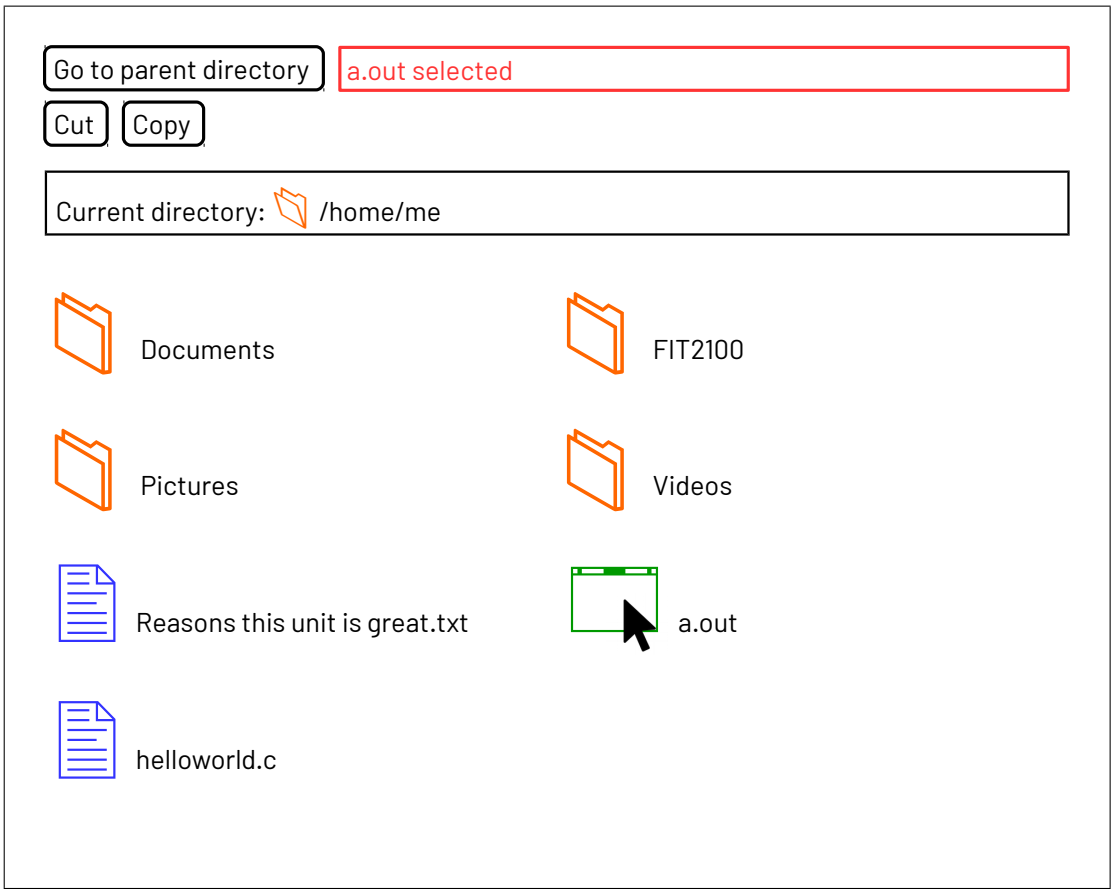

Figure 3: One possible way to show the user extra file commands for Task 3. In this example, the user has right-clicked `a.out`. The program responds by making extra buttons visible at the top of the screen, and uses a status box to remind the user of which file they selected. If the user clicks a different file without choosing any of the copy/paste options, the operation will be cancelled. Your implementation might be very different, but try to find a way to make it reasonably user-friendly.

Choose **any one** of the following feature sets to implement:

2.4.1 Option A: File properties

(a) Allow the user to view properties for a particular file. These should include file size,[5] owner of the file, and file permissions. AND...

(b) Allow the user to rename a file. The user should be asked to enter a new name for the existing file.

2.4.2 Option B: Cut, copy, paste

(a) Allow the user to copy a file[6] into a different directory location by implementing copy/paste commands (i.e. the user can select a file for copying, then navigate to the destination directory and select the paste option). If the user tries to place a copied file in a location where the same filename already exists, you should prepend '`copy of`' to the new filename to allow the operation to take place safely. AND...

(b) Allow the user to *move* a file to a different location by adding a cut command that works with your paste function. When the user selects `cut`, the original file should be designated for moving in some way, but should not actually be moved until the user selects `paste` in the new location.

2.4.3 Option C: Create and edit files

(a) Allow the user to create a new empty file. The user should be asked to enter a name for the new file. AND...

(b) Allow the user to open a file for editing (in addition to the default `less` viewing function), by opening the file in a suitable text editor. The user should be able to customise the command used to run their favourite editor by setting an `EDITOR` environment variable[7] before they start your shell program. Add instructions in your user documentation on how the user can set their preference for their preferred editor.

[5] You are not required to display size information for directories.

[6] For this task, you only need to allow the user to copy and move non-directory files.

[7] e.g. The user might type `export EDITOR=/usr/bin/pluma` into the bash prompt before starting your program, or they may have this set in their user profile so their favourite editor is set every time they log in.

Caution! If you implement certain file operations incorrectly, your program might accidentally move or overwrite the wrong files! Take extra care and program defensively to ensure your shell does not produce undefined behaviour.

2.5 Important Notes

Commenting your code is essential as part of the assessment criteria (refer to Section 2.6). You should also include comments at the beginning of your program file, which specify your name, your Student ID, the start date and the last modified date of the program, as well as with a high-level description of the program. In-line comments within the program are also part of the required documentation.

You are also required to produce a user manual explaining how to compile and run your graphical and non-graphical shell implementations. You should explain how the user can make use of each piece of functionality you have implemented, as well as any assumptions and limitations relevant to your implementation. You should include this document in plain-text so that the user can access it through your shell.

2.6 Marking Criteria

The assessment of this assignment will be based on the following marking criteria. The same marking criteria will be applied on all tasks:

- 50% for working programs (for all three tasks);
- 20% for code architecture (algorithms, use of procedures and libraries, etc. in your implementations of the three tasks)
- 10% for general coding style (clarity in variable names, function names, blocks of code clearly indented, etc.)
- 20% for documentation (both program comments and user documentation)

3 Submission

There will be NO hard copy submission required for this assignment. You are required to submit your assignment as a `.tar.gz` file name with your Student ID. For example, if your

Student ID is 12345678, you would submit a zipped file named `12345678_A1.tar.gz`. Note that marks will be deducted if this requirement is not strictly complied with.

Your submission is via the assignment submission link on the FIT2100 Moodle site by the deadline specified in Section 1, i.e. **24th August 2018 (Friday) by 5:00pm**.

3.1 Deliverables

Your submission should contain the following documents:

- A completed the assignment cover sheet for online submission available on the FIT2100 Moodle site.
- A user documentation file (not more than 3 pages) in plain .txt format with clear and complete instructions on how to run your programs. (Note that your programs must run in the Linux Virtual Machine environment which has been provided for this unit. Any implementation that does not run at all in this environment will receive no marks.)
- Electronic copies of ALL your files that are needed to compile and run your programs.

Marks will deducted for any of these requirements that are not strictly complied with.

3.2 Academic Integrity: Plagiarism and Collusion

Plagiarism Plagiarism means to take and use another person's ideas and or manner of expressing them and to pass them off as your own by failing to give appropriate acknowledgement. This includes materials sourced from the Internet, staff, other students, and from published and unpublished works.

Collusion Collusion means unauthorised collaboration on assessable work (written, oral, or practical) with other people. This occurs when you present group work as your own or as the work of another person. Collusion may be with another Monash student or with people or students external to the University. This applies to work assessed by Monash or another university.

It is your responsibility to make yourself familiar with the University's policies and procedures in the event of suspected breaches of academic integrity. (Note: Students will be asked to attend an interview should such a situation is detected.)

The University's policies are available at: `http://www.monash.edu/students/academic/policies/academic-integrity`