Curtin University — Department of Computing

Unix & C Programming (COMP1000)

Semester 1, 2022

Final Assessment

Due: Tuesday 14 June 2022, 11:59 PM (Perth Time)

Weight: 50% of the unit mark.

This is an online OPEN BOOK Assessment. You have 7 days after the release of this specification to complete this assessment (see above for the deadline). There are 2 main tasks on this assessment. The first one is the Bash Command task, while the second one is the major C programming task. Both of these tasks are independent of each other. Finish all the tasks and submit the ZIP file before the due date. Late/corrupted/draft submission will not be accepted. Please ensure that you dedicate sufficient time to understand the tasks at hand before you begin coding.

Academic Integrity

This is an assessable task, and as such there are strict rules. While the assessment is *open book*, this does not allow collusion. You must not ask for or accept help from anyone else on completing the tasks (whether or not they are themselves students). You must not show your work to another student enrolled in this unit who might gain unfair advantage from it. These actions are considered plagiarism or collusion.

Do NOT just copy some C source codes from internet resources. If you get caught, it will be considered plagiarism. Always reference your sources (on a text file). Be aware that if significant portions of the assignment are not your own work (even if referenced), there may be penalties. If in doubt, ask!

Please see Curtin's Academic Integrity website for information on academic misconduct (which includes plagiarism and collusion).

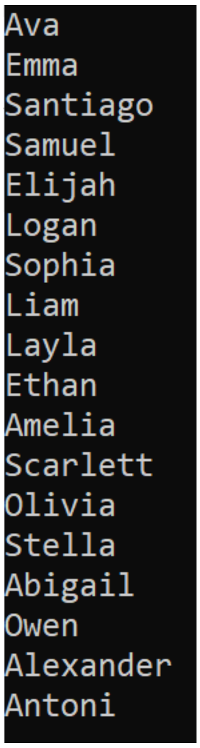
The unit coordinator may require you to attend quick interview to answer questions about any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an Academic Misconduct inquiry. In addition, your assignment submission maybe analysed by systems to detect plagiarism and/or collusion.

1 Bash Command

This is your first task of your final assessment. This task requires you to figure out the bash command needed to accomplish the tasks below. Please save your answer in the Shell Script format (in .sh format) so that we can execute the command.

You will be provided a text file called ”name.txt” that contains the list of various random names as below:

Figure 1: Text file containing names.



```
Ava
Emma
Santiago
Samuel
Elijah
Logan
Sophia
Liam
Layla
Ethan
Amelia
Scarlett
Olivia
Stella
Abigail
Owen
Alexander
Antoni
```

Using **grep**, **regular expression**, and **pipe**, write an executable Shell Script with a **single-line** command to execute each of the following task:

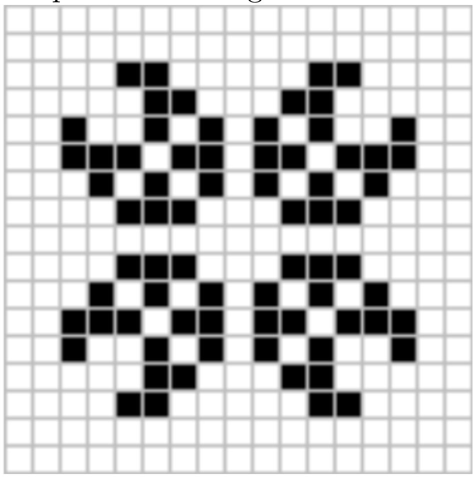
- Print all the names that do NOT contain vowel as the last letter. (Hint: use ”man grep” to learn the features). (Save your answer as ”q1.sh”)
- Print all the names that start with either ’A’ or ’B’, AND contains 5 letters or less. (Save your answer as ”q2.sh”)
- Print all the names that contains substring ”am” (case insensitive), and then sort the list alphabetically (ascending). (Hint: Do some research on which command can be used to sort list) (Save your answer as ”q3.sh”)

2 Game of Life

This is the second task of your final assessment. This task will take longer than the first task. It is strongly recommended for you to watch the supplementary videos prior to the coding. The task will be to write a program that visualizes **John Conway's Game of Life**. We will refer this as "Game of Life" onwards. Feel free to read the Wikipedia article about "Conway's Game of Life". We provide the summarized explanation as follows.

You can imagine "Game of Life" as a simple simulation with simple rules that requires NO interaction from the player. Imagine that you have a 2-D grids (like a 2D array). Each individual is called a "cell" with two possible states: **alive** or **dead**. Typically the dead cell is represented as bright/white coloured cell, while the alive cell is represented as dark/black coloured cell.

Figure 2: An example of the cell grid from the Wikipedia article.



Assuming that you have initial cell grid (such as figure 2), we can update this cell grid on the next cycle/iteration by observing the eight neighbours surrounding each individual cell (horizontal, vertical, and diagonal neighbour cells). Here are the rules list to update the cell grid:

- If the cell is **alive** AND there are only **ZERO** or **ONE** alive neighbour cells, this will cause the cell to be **dead** on the next cycle. (Underpopulation)
- If the cell is **alive** AND there are **TWO** or **THREE** alive neighbour cells, the cell will stay **alive** on the next cycle. (Just right)
- If the cell is **alive** AND there are **more than THREE** alive neighbour cells, this will cause the cell to be **dead** on the next cycle. (Overpopulation)
- If the cell is **dead** AND there are exactly **THREE** alive neighbour cells, the cell will be **alive** on the next cycle. Otherwise, it stays dead.

This is basically the whole game. You will implement the program to simulate the game of life. You can try testing various cell patterns from the Wikipedia article to check the correctness of your program.

2.1 Command Line Arguments

Your program need two command line arguments. This is the command format when you run the program:

./life <cell_file> <cycles>

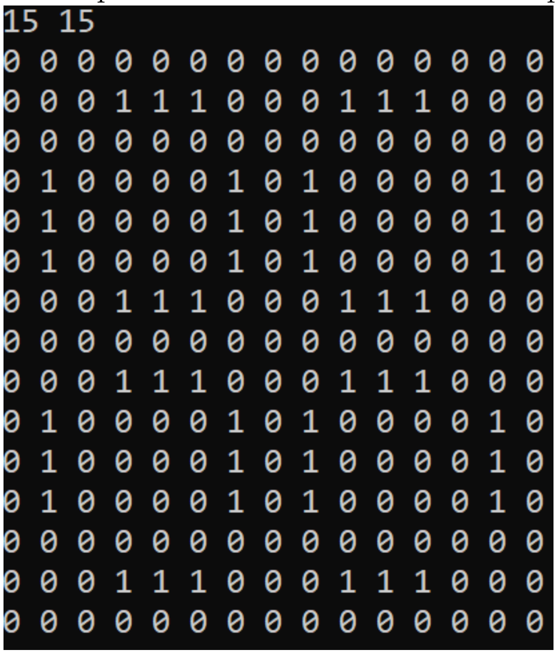
"./life" is the name of the executable. <cell_file> is the name of the text file containing the cell grid size and the initial cell state information. <cycles> is the amount of iterations/cycles your program will update the cell before it closes. Each iteration will have 1 second gap in between (using sleep() function).

If the user does not provide sufficient argument, the program should prompt the error message and ends the program. (Do NOT use exit() function). You can assume the second argument is always an integer. However, you need to make sure it is a positive integer.

2.2 Content of Text File

The program will attempt to open <cell_file> for reading the initial cell grid information. If the file opening fails, it should prompt an error message and end the program. (NOT with exit() function) Here is one example of the text file:

Figure 3: Sample text for Penta-decathlon cell pattern.



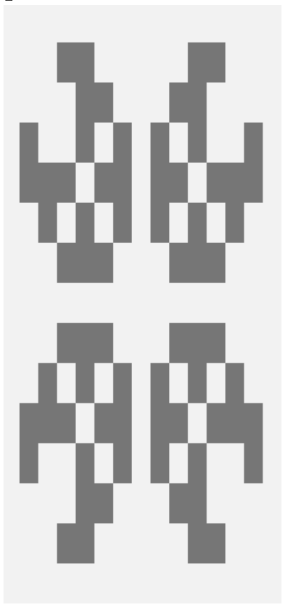
```
15 15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0
0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The first two integers on the first line are the row and column size of the cell grid respectively. The remaining integers are the content of the cell grids. These numbers represent the state for each cell. Integer 0 (zero) represents the dead cell, while integer 1 (one) represents the alive cell. Please use malloc() function to allocate proper 2D array to store the data. You can assume the cell state will always be either zero OR one in the text file. You can also assume the amount of the cell states always matches the cell grid size. (NO extra number OR lack of number)

2.3 Cell Grid Display

Your program should be able to display the content of the cell states with different background colors. Do NOT just print ’0’ and ’1’ on the terminal. Please watch the supplementary video for further explanation on the way to achieve this. Here is an example:

Figure 4: Display the cell states with background colors. Bright/while color represents dead cell, and Dark/black color represents alive cell.



You can print a single white space ’ ’ and change the background color for each cell. Please refer to file ”sampleSleepAndBackgroundColor.c” for guidance.

2.4 Cell Update on Each Cycle

Once you implement the cell grid display correctly, it is time to put them in a repeating cycle with the game rules applied. Please refer to section 2 for the list of the rules. Your program will update the cell grids as many as argument <cycles>. For each iteration, your program should clear the screen before displaying the cell grid, AND sleep for 1 second afterwards. We have provided various cell grid pattern text files to test your program.

2.5 Makefile

As usual, please write a proper makefile with all the variables, flags, and rules (including clean rules). Furthermore, please make sure all the rules should **only** have the relevant prerequisites. If this makefile is missing or broken, we will assume the program cannot be compiled. Keep in mind that an uncompilable program will have negative impact on the marking of your program (50% mark cap on the functionalities that do not work).

3 Marking Distribution

- **(15 marks)** Your Shell script works as intended. ”q1.sh” is 5 marks, ”q2.sh” is 5 marks, ”q3.sh” is 5 marks.
- **(5 marks)** Having a working proper makefile with proper flags and prerequisites.
- **(5 marks)** Proper structuring of the code with reasonable header files (with header guard). If you only have one .c file, then you get zero mark here.
- **(5 marks)** Good coding standard with C89 syntax. Any violation will result in zero mark on this category. Use the compilation flags ”-Wall -ansi -pedantic” to ensure your code complies with C89 standard.
- **(10 marks)** No Memory Leaks. The lack of malloc/calloc usage will give you zero marks here.
- **(10 marks)** Verifies the argc and ensure the command line argument is provided before continuing. <cycles> argument should NOT be non-negative number. If opening <cell_file> fails, the program should prompt error message and close the program. (do NOT use exit() function)
- **(15 marks)** Clear the terminal screen and display the cell grid with space character ’ ’ on correct background color.
- **(15 marks)** Can open text file and read the content correctly. 2D array should be malloc’ed based on the size provided in the text file. If you hardcode the array content manually, you will get zero marks here.
- **(15 marks)** The cell grids are updated correctly on every cycle based on the rule of the ”game of life”. Please test it on various patterns.
- **(5 marks)** The program sleep for 1 second on each cycle.

4 Final Check & Submission

After you complete your assignment, please make sure it can be compiled and run on our Linux lab environment. If you do your assignment on other environments (e.g on Windows operating system), then it is your responsibility to ensure that it works on the lab environment. In the case of incompatibility, it will be considered "not working" and some penalties will apply. You have to submit a **ZIP file** containing:

- **Your essential assessment files** — Submit all .c, .h, and .sh files along with your makefile. Please do not submit any executable or object (.o) files. Since there are 2 tasks (Bash Command & Game of Life), you can put them into 2 separate directories (for example, folders "task1" and "task2")
- **Declaration of Originality Form** — Please fill this form as well. Otherwise, your submission will not be marked.
- **Brief Report** — If your program is not perfect, please include a text file explaining the missing features. This will help us on the marking.

The name of the ZIP file should be in the format of:

$$\langle tutor-name\rangle\_\langle student-ID\rangle\_\langle your-full-name\rangle\_Final.zip$$

Example: Antoni-Liang_12345678_Arlen-Brower_Final.zip

Once the ZIP file is ready, please submit it on the Blackboard. Please make sure your submission is complete and not corrupted. You can re-download the submission and check if you can compile and run the program again. Corrupted or empty submissions will not receive marks. You may submit multiple times, but only your latest submission will be marked (assuming not over the due date). If your latest submission is a "Late" submission, you will receive zero mark regardless whether you have earlier submission or not. The files that are NOT inside the ZIP file will be ignored, so please make sure you ZIP every files into it. If you need to submit it multiple times, make sure the latest submission is still a complete submission (i.e not just one file to complement previous submission).

End of Final Assessment