MONASH University

FACULTY OF INFORMATION TECHNOLOGY

FIT1047 Introduction to computer systems, networks and security - S1 2022

Assignment 2 – Processes and MARIE Programming

Purpose	Processes and programs are what makes computers do what we want them to do. In the first part of this assignment, students will investigate the processes running on their computers. The second part is about programming in MARIE assembly language. This will allow students to demonstrate their comprehension of the fundamental way a processor works. The assignment relates to Unit Learning Outcomes 2, 3 and 4.
Your task	For part 1, you will write a short report describing the processes that are running on your computer. For part 2, you will implement a simple game in the MARIE assembly language.
Value	25% of your total marks for the unit The assignment is marked out of 50 marks.
Word Limit	See individual instructions
Due Date	**11:55 pm Thursday 14 April 2022**
Submission	● Via Moodle Assignment Submission. ● Turnitin and MOSS will be used for similarity checking of all submissions. ● This is an individual assignment (group work is not permitted). ● You will need to **explain your code** in an interview.
Assessment Criteria	Part 1 is assessed based on correctness and completeness of the descriptions. Part 2 is assessed based on correctness of the code, documentation/comments, and test cases. See instructions for details.
Late Penalties	● 10% deduction per calendar day or part thereof for up to one week ● Submissions more than 7 calendar days after the due date will receive a mark of zero (0) and no assessment feedback will be provided.
Support Resources	See Moodle Assessment page
Feedback	Feedback will be provided on student work via: ● general cohort performance ● specific student feedback ten working days post submission

INSTRUCTIONS
This assignment has two parts. Make sure you read the instructions carefully.

Part 1: Processes (10 marks)

For this task, write a brief report about the processes that are running on your computer. You can use one of the following tools (depending on your operating system):

- On Windows, use the Task Manager
- On macOS, use the Activity Monitor
- On Linux, use a command line tool like htop, or the ps command

Answer the following questions:

1. Briefly describe the columns displayed by the tool you use that relate to a) memory usage and b) CPU usage of a process. What can you say about the overall memory usage of all processes, compared to the RAM installed in your computer? (4 marks)
2. Pick a process you perhaps don't know much about, or which you did not expect to find running on your computer. Try to find out and describe briefly what it does. (4 marks)
3. Briefly explain why it is important that the operating system manages the files stored on your computer (rather than each application having to manage those files itself). (2 marks)

It may be useful to include a screenshot of your processes in the report. The word limit for this part (all three questions together) is 600 words (about 1 page, not including images).

Part 2: MARIE Programming (40 marks)

In this task you will develop a MARIE program that implements a clone of the popular Wordle game. We will break this task down into small steps for you.

Most of the tasks require you to write **code** and **test cases**. On Moodle, you will find a **template** for the code. It contains some subroutines that you should use, as well as a number of predefined test cases. **Your submission must be based on this template**, i.e., you must add implementations of your own subroutines into this template.
The code must contain comments, and you submit the `.mas` file together with the rest of your assignment.

In-class interviews: You will be required to demonstrate your code to your tutor after the submission deadline. Failure to demonstrate will lead to zero marks being awarded for the entire assignment.

Code similarity: We use tools to check for collaboration and copying between students. If you copy parts of your code from other students, or you let them copy parts of your code, you will receive 0 marks for the entire assignment.

Rubric: Each task below is worth a certain number of marks. A correctly working version of each task that is well documented and contains the required test cases will receive full marks. Missing/incomplete documentation will result in a loss of up to ¼ of the task's marks. Missing or undocumented test cases result in the loss of 1 mark per test case.

The Game

The goal of the game you're going to implement is that a player needs to guess a 5-letter word selected by the computer. The player has six attempts to guess. After each guess, the computer gives feedback for each letter: any letter that is correct is printed in upper case, any letter that appears in the word but in a different position is printed in lower case, and any letter that doesn't appear in the word is replaced with an underscore. If all letters are correct, or the player has exhausted their six guesses, the game ends.

Here is a typical game session. Let's assume the computer's word is SHIRE, but of course we don't know that yet:

Computer output:	`Enter your guesses`
My input:	`SPEAR`
Computer output:	`S_e_r`
My input:	`SHRED`
Computer output:	`SHre_`
My input:	`SHORE`
Computer output:	`SH_RE`
My input:	`SHIRE`
Computer output:	`SHIRE`

Strings

This section doesn't contain any tasks for you yet, it only introduces a concept you need for the rest of the assignment.

A *string* is a sequence of characters. It's the basic data structure for storing text in a computer. There are several different ways of representing a string in memory – e.g. you need to decide which character set to use (Unicode or ASCII), and how to deal with strings of arbitrary length.

For this assignment, we will use the following string representation:

- A string is represented in a contiguous block of memory, with each 16-bit memory location containing one character.
- The characters are encoded using the ASCII encoding.
- The end of the string is marked by the value 0.

As an example, this is how the string FIT1047 would be represented in memory (written as hexadecimal numbers):

```
046 049 054 031 030 034 037 000
```

Note that for a string with *n* characters, we need *n+1* words of memory in order to store the additional 0 that marks the end of the string.

In MARIE assembly, we can use the `HEX` or `DEC` keywords to initialise the MARIE memory with a fixed string:

```
FIT1047,     HEX 046
             HEX 049
             HEX 054
             HEX 031
             HEX 030
             HEX 034
             HEX 037
             HEX 000
```

After assembling this code, you can verify that the string has been loaded into the MARIE memory.

The assignment template contains two subroutines that you can use for this assignment. The `PrintString` subroutine outputs a string one character at a time using the `Output` instruction. The `InputString` subroutine reads a string one character at a time using the `Input` instruction.

Tip: you can switch the type of input in the MARIE simulator. For the actual string, use the Unicode input method so you can enter actual characters. For the final 0, switch to the decimal input method.

Task 2.1: Your name as a MARIE string (4 marks)

This is the first task you need to submit. It's just a little warm-up so you can get familiar with strings.

Similar to the FIT1047 example above, encode your name using ASCII characters. You should encode at least 10 characters – if your name is longer, you can shorten it if you want, if it's shorter, you need to add some characters (such as !?! or ..., or invent a middle name).

In the assignment template, you will find a label `Name`, which is initialised to the value `HEX 0`. You need to change this part of the code so that after assembling, the MARIE memory contains the string with your name starting at label `Name`.

Task 2.2: Convert to upper case (8 marks)

In this task, you need to write a subroutine that takes the user's input and converts it into upper case. The subroutine is called `ToUpper`. It expects one argument to be passed in using the label `ToUpperWord`, which is the starting address of the string to be converted.

The subroutine loops through the input string one character at a time. For each character, if it is a lowercase character, it replaces it with the corresponding uppercase character. Once the end of the string is reached, the subroutine returns.

Test cases: The template already contains one test case. Add another 3 test cases and document them. For each case, document which part of the subroutine it is supposed to test, as well as the expected outcome.

Task 2.3: Test if a word contains a certain character (10 marks)

Here, you will write a subroutine that can check if a word contains a given character. The subroutine is called `StringContains` and takes two arguments: the first is passed in the label `StringContainsStart`, and contains the address in memory where the string starts. The second is passed in the label `StringContainsChar`, which holds the character to search for. When the subroutine returns, it stores a 1 in the address at label `StringContainsResult` if the character was found, and a 0 if it wasn't found.

The subroutine should follow these steps:

- Loop through all the characters in the given input string.
- If the current character is the end-of-string character 0, then store 0 in `StringContainsResult` and return.
- If the current character is equal to the one in `StringContainsChar`, then store 1 in `StringContainsResult` and return.
- Otherwise, move on to the next character and continue the search.

Test cases: The template already contains one test case. Add another 4 test cases and document them. For each case, document which part of the subroutine it is supposed to test, as well as the expected outcome.

Task 2.4: Check a guess (10 marks)

In this task, you implement the core of the game. You need to write a subroutine `CheckGuess` that compares the player's input with the selected word and outputs the clues for the next round. The subroutine has two arguments. The first, which is passed in the label `CheckGuessedWord`, is the address in memory where the string starts that the player guessed. The second argument is passed in the label `CheckSelectedWord` and contains the address where the string of the selected word starts.

For simplicity, we will assume that both words only contain uppercase characters.

The algorithm for checking a guess needs to work as follows. Loop through both strings simultaneously, one character at a time:

- If the current character in the guessed string is 0 (i.e., the end of the string), check if the current character in the selected word is also 0. If yes, return from the subroutine. If not, output the error message "Your word is too short" and return from the subroutine.
- If the current character in the selected word is 0, but the current character in the guessed string is not 0, output the error message "Your word is too long" and return from the subroutine.
- If the current character in the guessed word is equal to the current character in the selected word, output the character (it will be an uppercase character, which indicates a correct guess).
- Otherwise, use your subroutine from Task 3 to test if the current character in the guessed word occurs somewhere else in the selected word
    - If yes, output the lowercase version of the character (since it's a good guess but in the wrong position).
    - If not, output an underscore character.

Test cases: The template already contains one test case, which calls the subroutine with **`GREAT`** as the guessed word and **`WORDS`** as the selected word. The output of your subroutine for this test case should be `_r___`, i.e., an underscore character (because `g` does not occur in `words`), followed by lower-case `r` (because `r` occurs but in a different position), followed by three underscores (because `e`, `a` and `t` do not occur in `words`). Add five more test cases and document them. For each case, document which part of the subroutine it is supposed to test, as well as the expected outcome.

Task 2.5: Main game loop (8 marks)

We can now use the subroutines to implement the game. The game itself will be implemented in a subroutine called Wordle, which has no arguments. The template already contains a subroutine that does nothing, so you need to fill in the remaining steps.

It should do the following:

1. Output a string to prompt the user to enter a guess
2. Get the guessed string from the user and convert it to upper case.
3. Use the CheckGuess subroutine to print the clues.
4. Check if the user has already completed 6 guesses. If yes, output a message that the game is over and halt. Otherwise, go back to step 1.

One thing we did not discuss above is how the computer selects its word. Usually, this would require a random number generator. Since that's too complex for this assignment, we will assume that the label `SelectedWord` is initialised statically with the start address of the selected word. (In the template, it's set to **`WORDS`** as in the test case.)

Test cases: Document 5 test cases. In this case, you are testing an interactive subroutine, i.e., one that expects user input. Your test cases are therefore not implemented as code, but written down as possible inputs and expected outputs.