MP3 LC3- Your-Own-Adventure (part 2)

Due Wednesday, Sept 17th, 2014 at 10:00 p.m.

Overview

As you know from last week, the program you are developing simulates a popular type of book in which the reader encounters specific choices while reading through the story. The reader must decide on a particular choice, then flip to the page corresponding to that choice to continue the adventure.

Building on your code from last week, you can now use the array form of the book to allow a player to play the game. The game implements a finite state machine for which the internal state is the current page being read by the player, and transitions occur in response to the player's choices.

Assuming that you successfully completed last week's program, this week's should require less effort.

Again, all routine details, such as how to obtain the code and how to hand in your program, can be found in the specification for MP1

The Pieces

This week, the starting file, **`prog3.asm`**, contains only a few strings that you will need. As your first step, you should copy your solution for MP2 into the **`prog3.asm`** file. We suggest that you then immediately commit that change.

On your own, you must write code to drive the choose-your-own-adventure game using the array form of the pages. **The game should begin only after your code from last week has been executed to create the array and print the array contents to the screen. If you implemented any of the error-checking challenges, and the database had errors, the program should simply stop after reporting the errors (without playing the game). If you implemented the path printing challenge for Program 2, turn path printing off before turning in this week's program.**

Details

Just as a reader must know the current page to continue reading a story, your program must keep track of the current page. The following paragraph describes how your program should function after creating and printing out the array (as in MP2):

Play begins on page 0. Print a message indicating that the game has started. Print the string for the player's current page. If the current page has no valid choices (all three entries in the array are -1), the game is over: print a termination message and HALT. If valid choices are available, print a prompt for player input. Then wait for the player to press a key using GETC and echo the key to the screen using PRINT CHAR (not OUT), followed by two ASCII linefeeds (0xA). The player's input is valid if the key pressed was between 1 and 3 and the corresponding choice (from the array) for the current page is not -1. If the player's input is not valid, print an error message and go back to re-print the string for the current page. If the player's input is valid, change the current page to the corresponding choice, and go back to print the string for the new page.

Example output based on **`test.asm`** is given below (and in the test files provided to you). The array printing output has been omitted; see the MP2 specification, or the test files provided to you, if you need to see an example of that part.

Example output (underlined characters are player input):

```
--- Starting adventure. ---

Story Text 0. Do you choose (1), (2) or (3)?
Please enter your choice: 3

Story Text 3. Do you choose (1) or (2)?
Please enter your choice: x



--- Invalid choice. ---

Story Text 3. Do you choose (1) or (2)?
Please enter your choice: 3



--- Invalid choice. ---

Story Text 3. Do you choose (1) or (2)?
Please enter your choice: 2

Story Text 4. Do you choose (1) or (2)?
Please enter your choice: 1

Ending Text 6.


--- Ending adventure. ---
```

Challenges for Program 3

Here are a few challenges for this week. Extra materials for testing challenge implementations are available in the challenge-materials subdirectory.

(3 points)
Use a stack to implement the ability to undo a player's choice. If the player presses the u key (lower case), use the stack to move them back to the previous page. Your program should still terminate when the player reaches a page with no valid choices (without allowing them to undo the choice that got them to that page). You must provide a stack of at least 512 memory locations, and may assume that a stack of that size will not overflow (we will not use such a test). You may **NOT** assume that the stack contains any previous pages (is not empty) when the player presses undo.

(10 points)
Implement word-wrapping for the page text (and not for any of the other strings printed to the monitor). The **`chained-nf.obj`** file contains a fully-reachable version of Surreal that use linefeeds only to end paragraphs. Rather than using PUTS to print the page strings, create and call a subroutine that inserts

additional linefeeds so as to limit the number of characters on each line to a fixed number L. The value L will be provided at memory location x0800; if L <= 0, you should simply use PUTS. Otherwise, your subroutine must place as many words from the string as can fit on each line, with at most L characters, and with gaps replaced by single spaces. Words are consecutive groups of non-space characters separated by one or more spaces or linefeeds (at paragraph breaks). Lines should be printed without leading or trailing spaces, which implies that any spaces between two words should be eliminated if your routine has chosen to end a line between those two words. Note also that if a single word is longer than L characters, your routine must break it across multiple lines.

(7 points) (REQUIRES PREVIOUS CHALLENGE)
Rather than simply wrapping the words, justify (align) the right edge of the text by evenly inserting extra spaces between words on each line other than the last in a paragraph. Extra spaces should be inserted starting from the right-most gap, so with five words (four gaps) and six more spaces needed, the first (leftmost) and second gaps should receive one extra space, while the third and fourth (rightmost) gaps receive two extra spaces. Words alone on a line should be left-justified.

Specifics:

- Your program must be written in LC-3 assembly language, and must be called **`prog3.asm`** — we will NOT grade files with any other name.
- Your code must begin at memory location x3000 (just don't change the code you're given in that sense).
- Database and array specifications are identical to those used with MP2, and all code corresponding to MP2 must meet the specifications for that assignment.
- The last instruction executed by your program must be a HALT (TRAP x25).
- You must use the PRINT CHAR subroutine to output single characters to the monitor (for example, when echoing the player's keystroke). You may use PUTS to print the string for each page.
- You will need to keep track of the current page. Play starts at page 0, and you should print P3_START_STR before play begins (but after printing the array, as in MP2).
- Before waiting for the player to press a key, your program should print the prompt at P3_PROMPT_STR.
- To access a particular record in your array, you can multiply the record index value by 4 to find its offset from x4000. You can then add 1, 2, or 3 to that offset to access its choice 1, 2, and 3 values.
- When -1 is stored in the array under the choice index field, your program must recognize that choice as invalid. You may assume that valid choices are listed first. For example, a case in which choices 1 and 3 are valid, but choice 2 is not, will never occur.
- An error message (given to you as P3_INVALID_STR) should be printed if the player types in an invalid choice or character at the prompt. Then the current page's string and input prompt should be printed again before waiting for the player to press another key.
- The end of the adventure is indicated by a -1 value in all three choice locations. Your program should then output the string P3 END STR and halt. Note that the adventure may end in different ways (at more than one location).
- Your output must match the desired format exactly, as shown in this specification and in the test files provided.
- You may not make assumptions about the initial contents of any register.
- Your code must be well-commented. Comments begin with a semicolon. Follow the commenting style of the code examples provided in class and in the textbook.

Testing

You should test your program thoroughly before handing in your solution. Remember, when testing your program, you need to set the relevant memory contents appropriately in the simulator. If you implemented the path printing challenge for MP2, turn path printing off before you test.

We have given you two sample test inputs, **`test.asm`** and **`surreal.obj`**. Note that you do not have the ASM version of surreal! ("Surreal" is a fairly extensive game in which you play a 198 student trying to finish your programming assignment, but to get to play the game, you have to finish your code...) You will need to assemble the **`test.asm`**. For each test, we have provided a script file to execute your program with the test input--**`runtest`** and **`runsurreal`**--and a correct version of the output: **`testout`** and **`surrealout`**. (We have also included **`chained.obj`**, **`runchained`**, and **`chainedout`** for those of you who test reachability.)

Remember to debug your code before trying the tests! And remember that your final register values need not match those of the output that we provide, but all other outputs must match exactly.

Grading Rubric:

Functionality (70%)

- 10% - program builds the correct array
- 10% - program prints the array values correctly
- 10% - player input is echoed to the screen
- 25% - player input is handled correctly (including invalid inputs)
- 15% - program prints starting, ending text, prompts, linefeeds correctly

Style (10%)

- 5% - necessary subroutines used for the MP2 portion
- 5% - clear code structure (portions for MP2 and 3 clearly defined)

Comments, clarity, and write-up (20%)

- 5% - introductory paragraph (extending that of MP2)
- 5% - table of registers included for **both old and new** sections
- 10% - code is clear and well-commented

Note that some point categories in the rubric may depend on other categories. Also note that if you were not able to complete last week's requirements, you can still get some points this week. To help those who were unable to finish MP2, we have provided the subroutine called PRINT_FOUR_HEX which prints four one-byte hex digits to the screen separated by spaces. Note that if you use this provided subroutine, the registers used as inputs for this subroutine may not match those in your code. However, the subroutine, if used as specified, will function correctly so be sure to match the registers as needed. Those points are free if you did complete the requirements...so long as you do not manage to break your solution, of course!