CIT 593 Homework 13: Dynamic Memory

Due Date: Tuesday 12/9/2014 @11:59pm via canvas upload ONLY

Note: Because LC-4's C doesn't have support for malloc & free, we will once again switch to a more mature OS that has the standard C memory related functions. The book will properly discuss these functions in ***Chapter 19*** *of the book.* While these more mature functions may operate differently internally on different OS's, they all behave the same way across the C-language. So the book is definitely a good source for learning about dynamic memory!

Assigned Problems (to be done individually, NOT group work):

For this assignment you will need to work & compile your C-programs on the university's server: eniac.

1) OVERVIEW: Implementing the PennSimm "loader" using Linked Lists

Overview: In the last HW, you wrote a program to open up a binary .OBJ file from PennSim and count the CODE/DATA/LABEL headers. In this problem, you will now interrogate the bodies following the headers and load that information into a linked-list that will mimic PennSim's program and data memory.

INPUT FILE FORMAT:

The following is the format for the binary .OBJ files created by PennSim from your .ASM files. It represents the contents of memory (both program and data) for your assembled LC-4 Assembly programs. In a .OBJ file, there are 3 basic sections indicated by 3 header "types" = CODE, DATA, SYMBOL.

- *Code:* 3-word header (xCADE, <address>, <n>), n-word body comprising the instructions. This corresponds to the .CODE directive in assembly.
- *Data:* 3-word header (xDADA, <address>, <n>), n-word body comprising the initial data values. This corresponds to the .DATA directive in assembly.
- *Symbol:* 3-word header (xC3B7, <address>, <n>), n-character body comprising the symbol string. Note, each character in the file is 1 byte, not 2. There is no null terminator. Each symbol is its own section. These are generated when you create labels (such as "END") in assembly.

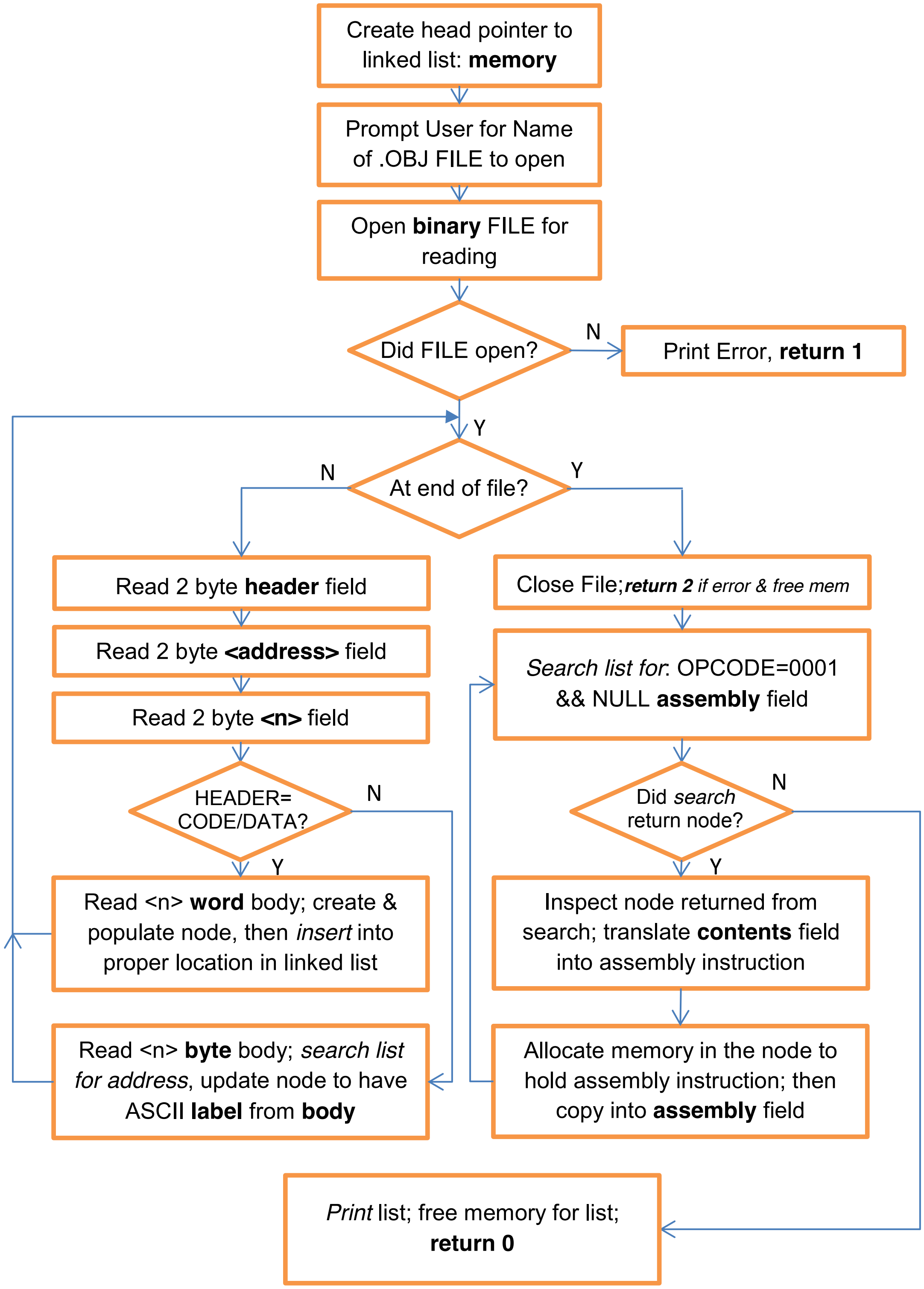
LINKED LIST NODE STRUCTURE:

The following structure (notice it has 5 **fields**) will be referenced in the problem description:

```
struct row_of_memory {
      short unsigned int address ;
      char * label ;
      short unsigned int contents ;
      char * assembly ;
      struct row_of_memory *next ;
} ;
```

CIT 593 Homework 13: Dynamic Memory

FLOW CHART: Overview of Program Operation



CIT 593 Homework 13: Dynamic Memory

IMPLEMENATION DETAILS:

For this HW, you will create a file called: **lc4_memory.c**. In this file you will create the structure that represents a row_of_memory as referenced above (see the section: LINKED_LIST_NODE_STRUCTURE above for the node's layout). You will also need to create several functions to create and manage a linked list of "rows_of_memory" nodes. The included files with the HW will help you get started with what functions to define.

Next, you will create a file called: **lc4.c** It will have only 1 function: **main()**. In main() you will create a head pointer to the linked list type you defined in lc4_memory.c, called: **memory**; that will hold the contents of program & data memory for your LC-4.

The last file you will create for this part will be called: lc4_loader.c. In this file, write a function to prompt the user for the name of the .OBJ "input" file and open the file in **binary** format for reading. If successfully opened; return a pointer to that open file.

Also in lc4_loader.c, write a second function to parse the .OBJ file. The format of the .OBJ input file is identical to your last HW, but its layout if reprinted above (see section: INPUT_FILE_FORMAT). As shown in the flowchart above, have the function read the 3-word header: header type, <address>, <n>. Determine the type of header you have read in: CODE/DATA/SYMBOL. Read in the address field, and the <n> field into local variables. If you encounter a CODE header in the .OBJ file, from the file format for a .OBJ file, we realize the body of the CODE section is <n>-words long. For the example of a CODE section printed below, we see **n**=0x000C, or decimal: 12. This indicates that the next 12-words in the .OBJ file are in fact 12 LC-4 instructions. Recall each instruction in LC4 is 1 word long.

```
CA DE 00 00 00 0C 90 00 D1 40 92 00 94 0A 25 00 0C 0C 66 00 48
01 72 00 10 21 14 BF 0F F8
```

From the example above, we see that the first LC-4 instruction in the 12-word body is: 9000. (that happens to be a CONST assembly instruction if you convert to binary). Allocate memory for a new node in your linked list to correspond to the first instruction (the section above: **LINKED LIST NODE STRUCTURE,** declares a structure that will serve as a blue-print for all your linked list nodes called: "row_of_memory"). As it is the first instruction in the body, and the address has been listed as 0000, you would populate the row_of_memory structure as follows.

address	0000
label	NULL
contents	9000
assembly	NULL
next	NULL

CIT 593 Homework 13: Dynamic Memory

In a loop, read in the remaining instructions from the .OBJ file; allocate memory for a corresponding **row_of_memory** node for each instruction. As you create each **row_of_memory** add these nodes to your linked list (you should use the functions you've created in lc4_memory.c to help you with this). For the first 3 instructions listed in the sample above, your linked list would look like this:

Header pointer: **memory**

address	0000
label	NULL
contents	9000
assembly	NULL
next	

address	0001
label	NULL
contents	D140
assembly	NULL
next	

address	0002
label	NULL
contents	9200
assembly	NULL
next	

Next node…

The procedure for reading in the DATA sections would be identical to reading in the CODE sections. These would become part of the same linked list, as we remember PROGRAM and DATA are all in one "memory" on the LC-4, they just have different addresses.

For the following SYMBOL header/body:

C3 B7 00 00 00 04 49 4E 49 54

The address field is: 0x0000. The symbol field itself is: 0x**0004** bytes long. The next 4 bytes: `49 4E 49 54` are ASCII for: `INIT`. This means that the label for address: 0000 is INIT. Your program must search the linked list: memory, find the appropriate address that this label is referring to and populate the "label" field for the node. Note: the field: **<n>** tells us exactly how much memory to malloc() to hold the string. 4 bytes in the case of: `INIT`. For the example above, the node: 0000 in your linked list, would be updated as follows:

address	0000
label	**INIT**
contents	9000
assembly	NULL
next	

Once you have read the entire file; created and added the corresponding nodes to your linked list, close the file and return to main(). If you encounter an error in closing the file, before exiting, print an error, but also free() all the memory associated with the linked list prior to exiting the program.

CIT 593 Homework 13: Dynamic Memory

2) Implementing a Reverse Assembler

In a new file: **lc4_disassembler.c**: write a third function that will take as input the populated "memory" linked list that contains the .OBJ's contents and translate the hex representation of some instructions into their assembly equivalent. You will need to reference the LC4's ISA to author this function. To simplify this problem a little, you don't need to translate every single HEX instruction into its assembly equivalent. Only translate instructions with the OPCODE: 0001 (ADD REG, MUL, SUB, DIV, ADD IMM)

As shown in the flowchart, this function will call your linked list's "search_by_opcode()" helper function. Your search_by_opcode() function should take as input an OPCODE and return the first node in the linked list that matches the OPCODE passed in, but also has a NULL assembly field. When/if a node in your linked list is returned, you'll need to examine the "contents" field of the node and translate the instruction into its assembly equivalent. Once you have translated the contents filed into its ASCII Assembly equivalent, allocate memory for and store this as string in the "assembly' field of the node. Repeat this process until all the nodes in the linked list with an OPCODE=0001 have their assembly fields properly translated.

As an example, the figure below shows a node on your list that has been "found" and returned when the search_by_opcode() function was called. From the contents field, we can see that the HEX code: 128B is 0001 001 010 001 011 in binary. From the ISA, we realize the sub-opcode reveals that this is actually a MULTIPLY instruction. We can then generate the string **MUL R1, R2, R3** and store it back in the node in the assembly field. For this work, I strongly encourage you to investigate the **switch()** statement in C (any good book on C will help you understand how this works and why it is more practical than multiple if/else/else/else statements). *I also remind you that you must allocate memory strings before calling strcpy()!*

NODE BEFORE UPDATE

address	0009
label	NULL
contents	128B
assembly	NULL
next	

NODE AFTER UPDATE

address	0009
label	NULL
contents	128B
assembly	**MUL R1, R2, R3**
next	

CIT 593 Homework 13: Dynamic Memory

Putting it all together:

As you may have realized main() should do only 3 things: 1) create and hold the pointer to your memory linked list. 2) Call the parsing function in lc4_loader.c. 3) Call the disassembling function in lc4_dissassembler.c. One last thing to do in main() is to call a function to print the contents of your linked list to the screen. Call the print_list() function In lc4_memory.c; you will need to implement the printing helper function to display the contents of your lc4's memory list like this:

```
<label>      <address>      <contents>      <assembly>
INIT         0000           9000
             0001           D140
             0002           9200
             …
             0009           128B            MUL R1, R2, R3
(and so on…)
```

Several things to note: There can be multiple CODE/DATA/SYMBOL sections in one .OBJ file. If there is more than one CODE section in a file, there is no guarantee that they are in order in terms of the address. In the file shown above, the CODE section starting at address 0000, came before the CODE section starting at address: 0010; there is no guarantee that this will always happen, your code must be able to handle that variation. Also, SYMBOL sections can come before CODE sections! What all of this means is that before one creates/allocates memory for a new node in the memory list, one should "search" the list to make certain it does not already exist. If it exists, update it, if not, create it and add it to the list!

Prior to exiting your program, you must properly "free" any memory that you allocated. We will be using a memory checking program known as valgrind to ensure your code properly releases all memory allocated on the heap! You are more than welcome to use this tool as well. Simply run your program: lc4_loader as follows:

```
valgrind lc4_loader
```

<u>STRUCTURING YOUR CODE:</u>

For ease of coding and for grading, use the following file names for your code:

`lc4.c`	- must contain your main() function.
`lc4_memory.c`	- must contain your linked list helper functions.
`lc4_memory.h`	- must contain the declaration of your <u>row_of_memory</u> structure & linked list helper functions
`lc4_loader.c`	- must contain your .OBJ parsing function.
`lc4_disasembler.c`	- must contain your disassembling function.
`Makefile`	- must contain an "all" and "clean" directives

3) EXTRA CREDIT: A complete reverse assembler:

Finish the disassembler to translate any/all instructions in the ISA. Have you program print the linked list to the screen still, but also create a new output file: <users_input>.asm. In that file it should contain only the assembly program that you disassembled. If it works, I should be able to load it into PennSim , assemble it, and reproduce the identical .OBJ file that your .ASM fileas derived from! Don't forget to add in the directives (.CODE, .DATA)…the ultimate test of your program will be getting it to assemble using PennSim!

CIT 593 Homework 13: Dynamic Memory

Directions on how to submit your work:

- Create a single zip file called: **LAST_FIRST_HW13.zip**
- The zip file should contain the 8 files named in this assignment.
- There should not be any sub-directories within your zip file.

As an example, I would turn in the following SINGLE zip file:

FARMER_THOMAS_HW13.zip

This single zip file would contain 8 files only*:*

lc4.c
lc4_loader.h
lc4_loader.c
lc4_memory.h
lc4_memory.c
lc4_disassembler.h
lc4_disassembler.c
Makefile

You will then upload ONLY 1 file to canvas: **FARMER_THOMAS_HW13.zip**

- DO NOT TURN IN ANY files **created by the compiler**, **.obj**, or **PennSim.jar**

- Make certain that you submit the latest version of your code.

- Submitting using any other compression type (.RAR, TAR, GZIP) will be rejected.

Paper/Email submissions will not be accepted for this assignment.