Computer Systems Organization
CSCI-UA.0201-005 Fall 2024

Programming Assignment 3
Due Friday, November 22 at 11:55pm

In this assignment, you will be writing assembly code corresponding to a C program that you are being provided. You will need to devote a large amount of time to this assignment, so please get started right away!

To complete the assignment, perform the following steps.

1. Download one of the following files attached to this assignment, according to the system you are using:
    - assignment3_macos.tgz for macOS
    - assignment3_linux.tgz for Linux
    - assignment3_cygwin.tgz for Windows/Cygwin

    Save and uncompress the downloaded file in the directory where you want to work on and compile your program. To uncompress the file, in a shell, type

    ```
    tar -xzvf filename
    ```

    where *filename* is the name of the file that you downloaded.

    The three files that are extracted from the compressed file are:
    - heap.c, a C file.
    - assignment3.s, an assembly language file where you will be putting your code.
    - makefile, a file that makes compiling quick and easy (do not change this file).
    - input.txt, a file (similar to the one for assignment 1) containing the input for the program.

2. Review the comments and C code in heap.c carefully. You should understand what is going on in that file. Be sure that you can compile and run the C code, by typing "make" (which will generate an executable named assign3, or assign3.exe in Cygwin) and then typing "./assign3 < input.txt". Since the instructions below ask you to modify heap.c, I suggest saving an unmodified copy of it, which you can compile and run to compare its output with that of your code.

3. You can see the definition of the RECORD struct type in heap.c. For your assembly programming, you will need to determine the size of a RECORD struct and the offsets of the fields of the struct. To do this, use C code similar to the file offset.c that I uploaded to Brightspace under the "Programs Discussed in Class" tab for Lecture 18. You can put

this code in the **`main`** function in heap.c, if you like, or write small, separate program to do this.

4. In heap.c, you will see two lines:
   ```
   int heap_count = 0;
   // extern int heap_count; // in assembly
   ```
   Comment out the first line and uncomment the second line, so that there is only the extern declaration. Then, in assignment3.s, add a declaration of a global variable **`heap_count`** that is 32 bits and whose initial value is 0. To see how to do this, look at the bottom of the assembly file, str_glob_xxx.s, on Brightspace under the "Programs Discussed in Class" tab for Lecture 18. Be sure to include the ".data" (indicating the data section) and ".globl" declarations in the assembly code (and, on macOS, put a "_" in front of heap_count). I recommend inserting this assembly code at the bottom of assignment3.s, but it's not necessary that it be at the bottom.

   Now, compile your program using "make" and run it, exactly as above.

5. In heap.c, you'll see the prototype for the function **`heap_swap`**, followed by the actual function definition. Comment out the actual function definition, leaving just the prototype. Then, in assignment3.s, write the assembly code corresponding to the **`heap_swap`** code that you commented out. This is an easy one, to give you a gentle start on the assembly code. I suggest the following:
   - You'll see in assignment3.s the skeleton of a definition of the function **`NAME`** (or **`_NAME`** for macOS). Change the name to **`heap_swap`** (or **`_heap_swap`** on macOS). Note that I put the calling convention information you need into the file.
   - I discussed in class how to access in assembly a global variable (in this case, the **`heap`** array variable) that is defined in C, and I put an example into the sample assembly code in sr_glob_xxx.s for Lecture 18. Note that if you want to have a register point to the heap array, you will need to use the "leaq" (load effective address) instruction to write the address of the heap into the register.
   - Note that each element of the **`heap`** array is a pointer, so is 8 bytes. You'll need to remember this when using indexed addressing to access the elements of the heap array.

   Once you have completed the assembly code for **`heap_swap`**, compile your program using "make" and run it the same way as above.

   Please note that this is by far the easiest assembly function you will be writing, so if you get stuck, you should seek help <u>immediately</u> by attending an office hour, emailing your tutor, or posting on the discussion board for Programming Assignment 3 – but remember not to post your own code.

6. In heap.c, you'll see the prototype for the function **`read_record`**, followed by the actual function definition. Comment out the actual function definition, leaving just the

prototype. Then, in assignment3.s, write the assembly code corresponding to the **`read_record`** code that you commented out. I suggest the following:

- Write the code for read_record under the code for heap_swap, above. Be sure to include the ".globl" declaration for read_record (remembering to use a "_" on macOS). See the Lecture 18 sample_xxx.s code as an example of having multiple assembly functions in a file.
- Because **`read_record`** makes lots of function calls (scanf, malloc, strcpy, etc.), I recommend using callee-saved registers, so that you don't have to save caller-saved registers before every call and restore them after every call. That is, at the beginning of the function (after the "movq %rsp,%rbp"), push the callee-saved registers that you want to use. Then, at the end of the function (before the "popq %rbp"), restore those callee-saved registers by popping them off the stack in reverse order of the pushes. Each push allocates 8 bytes on the stack – be sure that you maintain 16-byte stack alignment by either doing an even number of pushes or, if there are an odd number of pushes, subtracting an additional 8 from the %rsp (and adding 8 back later on).
- The local variables **`lastname`**, **`firstname`**, and **`id`** should be allocated on the stack, as discussed in class. You can use a register to hold the variable **`p`**, if you like.
- When you need to get the address of something, be sure to use the "leaq" (load effective address) instruction. For example, the call to scanf,
  ```
       int res = scanf("%s %s %ld", lastname, firstname, &id);
  ```
  needs to provide the addresses of the format string, **`lastname`**, **`firstname`**, and **`id`**. Thus, you will need to use leaq when writing these addresses into registers to pass them to scanf.
- The above format string (and the format strings for printf elsewhere in the program) should be defined as discussed in class and seen in the str_glob_xxx.s file for Lecture 18, using ".asciz".
- You will need to pass the size of the RECORD struct (above) to malloc and you will need to use the offsets of the fields of a RECORD (also above) to access those fields.

As usual, once you have completed the assembly code for **`read_record`**, compile your program using "make" and run it the same way as above.

7. The next function to implement in assembly is **`sift_up`**. Comment out the code for **`sift_up`** in heap.c, leaving the prototype. Even though the C code uses the **`PARENT`** macro, you'll need to write the corresponding assembly code (which is just subtracting one and shifting right) instead. Don't bother to make **`PARENT`** a function on its own.

8. Continue with the same process of commenting out the C code and writing in assembly the definitions of the functions **`sift_down`**, **`heap_insert`**, and **`heap_remove`**. The **`sift_down`** function will require concentration on your part, the other two are easy. Do not implement the **`main`** function in assembly, leave it in heap.c.

9. When you are finished and the code is working correctly, just upload the assignment3.s file to Brightspace. Do not upload heap.c.

Important: Debugging assembly is challenging. I tend to use print statements for debugging, which you can do in assembly by calling printf. I find that my programs often crash (with a "segmentation fault") because I have not maintained the stack properly (e.g. decrementing the stack pointer and forgetting to increment it again), have forgotten to maintain a 16-byte alignment for the stack pointer, have forgotten to save callee-saved registers before using them, or have forgotten to save the caller-saved registers that I am using before calling another function.