CS 3113: Operating Systems

Spring 2022

Dr. Richard M. Veras

Midterm

Due: March 27th, Bonus March 20th

Goal: In this exam you are to implement two programs and answer questions about the programs and various OS topics. The answers to the questions are not necessarily in the lectures, labs, homework, or readings. Instead, in answering the questions, you will bridge the topics covered in these facets of the class. You will need to do some technical-googling (an important skill), outside reading, and critical thinking to answer these questions. The hints provided will help get you started with both the code and the questions.

Deliverables: to be uploaded to canvas

1. An assembly file **fb.s** that implements fizzbuzz
2. A C file **list.c** that implements the linked list program.
3. Your own test cases for the linked list program: **sample.input**
4. And the expected results from your tests cases: **expected.output**
5. A PDF file **report.pdf** that contains your answers to the questions.

Rules: This is an open book take-home exam. You can use any resource, **except each other**, and **you must** cite your sources. If you leverage existing code, then **you must** make abundantly clear where that code came from and what changes you made.

Grading: Each correct program is worth 20 points for a total of 40 points. Each question is worth 4 points each for a total of 60 points. If the non-code components (report.pdf, sample.input, expected.output) are submitted at least **1 week before the deadline then you will get a 10-point bonus**. Any substantial changes to the non-code components after 1 week prior to the deadline forfeits the bonus. The maximum number of points for this exam, including the bonus, is 100 points.

FizzBuzz in Assembly

Context: FizzBuzz is a software engineering interview question and the textbook example used in teaching Test Driven Development (TDD) techniques, such as Red, Green, Refactor. For technical interviews it is used as a way for the interviewer to see how you think and reason through the design of a program starting from the specification. I am adding an extra level of complexity by having you do this in assembly, so do not feel discouraged. This is meant to be hard.

Problem: In x86_64 (or x86) assembly create a program that loops through the integers 1 through 100. For each value **x** print out a line that is either the string "Fizz" if **x** is divisible by 3, "Buzz" if it is divisible by 5, or "FizzBuzz" if it is divisible by both 3 and 5. Do not hardcode the outputs.

Input: <None>

Output: 100 lines of either an integer, “Fizz”, “Buzz”, or “FizzBuzz”

Example:

Input	Output
	1 2 Fizz 4 Buzz Fizz 7 … <snip> … 14 FizzBuzz 16 ... Buzz

Hints:

1. Prototype this in a language you are most comfortable with, then write it in assembly language.
2. Break this up into two parts. First figure out the control flow of looping from 1-100 and using conditionals. Second, figure out how to use printf in assembly language. Then combine the two.
3. Printing with printf, get at least one of these to work: this, this, but different syntax
4. Linking against the C library. To use **printf** you will need to add an extra flag to **ld**:

```
as fb.s -o fb.o
ld -melf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 fb.o -lc -o fb.x
```

Questions:

1. Explain in words how your implementation works
    a. What were your assumptions? How did you interpret the specification?
    b. What state do you need to store? In memory? In Registers?

    c. How does your algorithm work?
    d. How do you plan to test your code?
    e. How do you plan to build your code?
2. What is the difference between x86 and x86_64 assembly?
3. How is assembly code translated into a binary that the OS can execute?
4. What is the general process for calling a function in assembly (x86_64)? How would you go about calling the function **printf** in assembly?
5. What is a system call? What happens when one is made? How is it different than a function call?
6. What is **glibc**, and why do we care about it? How is the functionality that it provides different from the OS system calls? How does it interact with OS system calls?

Doubly Linked List Interpreter

Context: Node-based data structures are the building blocks of complex data structures, such as linked-lists, queues, trees, tables and graphs. Many important data structures that you will come across in operating systems will be a node-based data structure. For example: Queues are used in scheduling processes to facilitate efficient virtualization of the processor. Doubly Linked Lists are the underlying structure in many memory allocators (malloc). Tables are the basis of traditional file systems and tree-linked-list-hybrids (btrees) are the basis of modern file systems.

In this program you will implement part of a doubly linked-list psuedo-database system. This program will force you to think about how to write system programs in C, how to manipulate command line data, how to read and write files, how to use structures and pointers and managing data on the heap with malloc and free.

Problem: Write a program that interprets commands from an input file to add, remove and print the elements of a double linked list. The command “addtail <key>” inserts a new node, with the value <key> at the tail of the list. The command “addhead <key>” inserts a new node with the value <key> at the head of the list. “del <key>” removes all nodes with the value <key>. And the command “print” writes the current state of the list to the output file. “<key>” can be any string except the strings “EMPTY”, “HEAD” and “TAIL”, these are reserved.

Input: The program will receive two command line arguments, an input file, and an output file. Both files are text files. The input file is structured as follows: The first line is an integer specifying the number of lines that will follow. Each line after that will take three different forms:

addtail <key>

addhead <key>

del <key>

print

Output: The program will output the results in an output text file specified in the command line arguments.

Each line in the output file will correspond to the results of the print statements in the input file.

The output of print takes the form “(k1,p1,n1), (k2,p2,n2), …., (km,pm,nm)”, where kX, pX, and nX are the keys of the current node, the key of the previous node and the key of the next node, respectively.

Example: The program “list.x” is run on the command line with two arguments, the name of the input file and the name of the output file.

./list.x test1.input test1.output	
Contents of **test1.input**	Contents of **test1.output** after running list
print addtail “a” print addtail “bb” print addtail “cd” print	EMPTY (a,HEAD,TAIL) (a,HEAD,bb),(bb,a,TAIL) (a,HEAD,bb),(bb,a,cd),(cd,bb,TAIL)

./list.x test2.input test2.output	
Contents of **test2.input**	Contents of **test2.output** after running list
addtail “a” print addhead “bb” print	(a,HEAD,TAIL) (bb,HEAD,a),(a,bb,TAIL)

./list.x test3.input test3.output	
Contents of **test3.input**	Contents of **test3.output** after running list
addtail “a” addhead “bb” print addhead “cab” del “bb” print	(bb,HEAD,a),(a,bb,TAIL) (cab,HEAD,a),(a,cab,TAIL)

Hints: Here are some ideas and examples. These are not the only ones, so do some searching.

1. Before coding, break this problem up into pieces for example:

    a. Reading the file names from the command line arguments. Here, here, here
    b. Opening and reading the input file and writing to an output file. Here, here, here. Also, fprintf, sprintf and company.
    c. Parsing the input commands as strings. Here, here and here. Also, strtok
    d. Dispatching to the appropriate function from the commands. here, here
    e. Recursive functions for adding nodes, deleting nodes and printing nodes. See lecture notes. Also, for ideas check here, here (java example), here (also not C)
2. Prototype in the language you are most comfortable with first.
3. Test those pieces individually.
4. Create test cases that exercise your code. (I.e. all branches are taken by your test cases)
5. GCOV can help with checking if your tests cover your code. Check out here, here and here.
6. Check out the command valgrind and use it to find memory leaks (not using frees) and bad memory accesses. Here, here, here and here.
7. Consider writing a makefile to build your code and run gcov and valgrind. Here, here and here.

Questions: You will need to research some of these questions, but you should have enough information from the course material to seek out these answers.

1. Explain in words how your implementation works
    a. What were your assumptions? How did you interpret the specification?
    b. What state do you need to store?
    c. What do your data structures look like?

    d. What are the pieces of your code and how do they work together?
    e. How does your algorithm work?
    f. How do you plan to test your code?
    g. How do you plan to build your code?
2. Prior to implementing your code, write an example input file (sample.input) and an output file with the expected data. Make sure these test cases are different from the ones provided and that they completely cover your code (not all branch will be taken by one individual cases, but the aggregate of them should exercise every line of your code).
3. How does Linux load a program from the file system and turn it to process? What are all the steps between?
4. What is the exec function? How does it work and why do we normally use fork and wait with it?
5. How does a C program in Linux get its arguments from the command line?
6. How is a file read and written to? What are the steps from the program side (start to finish how do you read and write a file in C)? From the OS perspective?
7. There are many special files for example there are the special files you have seen STDIN, STDOUT, and STDERR, and there are others that you have not yet seen, such as Unix Pipes and Unix Sockets. What are these special files, why would you use them and how?
8. What are pointers in C? What can we not do without them? How do they relate to the things we have done in assembly?
9. What are the stack and heap? How are they similar and how are they different? From start to finish, how are they used in C?