Rebekah Lee

PART 1

Questions:

1. Explain in words how your implementation works

a. What were your assumptions? How did you interpret the specification?

I will start with an index register %rdi starting from 1 and created a loop that increments %rdi by 1 until it reaches 100. I will also add the arithmetic portion inside the loop. I will use %rbx to store the current number and based on the given math problem, it will print "Fizz" if the number is divisible by 3 and store it in %rax, "Buzz" if it is divisible by 5 and store it in %rcx, or "FizzBuzz" if it is divisible by both 3 and 5 and store it in %rdx using printf instruction.

Since there are no inputs, my assumptions are that the code will automatically run by printing off numbers from 1 to 100 and change to either Fizz, Buzz, or FizzBuzz according to the given instructions in the loop.

b. What state do you need to store? In memory? In Registers?

I will use %rdi for index counter, %rbx to store current number, %rax to store any number that's divisible by 3, %rcx to store any number divisible by 5, and %rdx to store any number that's divisible by both 3 and 5.

c. How does your algorithm work?

As the index counter increases by 1 in the loop, the current number also increases by 1 and it will do a math to see if the current number is divisible by 3, 5, or both. If it is divisible by 3, 5, or both, then it will print out the words according to the instructions.

d. How do you plan to test your code?

I plan to test my code by seeing the output every time I make a change inside the loop. In the beginning, I plan to print if my numbers are correctly incrementing from 1 to 100. Then, I will test to see if the registers are correctly saving the numbers. Then, I will apply the math portion by first checking if it will print "Fizz" if the number in the given range is divisible by 3. Once that passes, I will check if it prints "Buzz"

if the number is divisible by 5. Then will check for “FizzBuzz” if divisible by both 3 and 5.

e. How do you plan to build your code?

I will first start with an index counter then a loop that does the math and prints then make the loop to repeat until it reached 100. I don’t think I will need another loop to do the math portion but if the code gets long, I might make another loop.

2. What is the difference between x86 and x86_64 assembly?

x86 only uses a 32-bit assembly code while x86_64 is a 64-bit assembly code.

3. How is assembly code translated into a binary that the OS can execute?

An assembler takes the assembly code and translates into a binary. We also call the translated binary version as an “Object File”. Then it links the files and get loaded into memory and be executed.

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?

You can use a stack to call a function in assembly. Instructions such as pushl, popl will work with registers and you can return with a ret instruction. Or you can use movq and call printf. You can also use the instruction call to call a function.

To call a function printf in assembly, you first need to create a word/sentence you would like to print in quotation marks and store it under a name. Then, you can movq the name and call printf in the end to print it out.

5. What is a system call? What happens when one is made? How is it different than a function call?

Cite: https://www.geeksforgeeks.org/introduction-of-system-call/

A system call is a way for a program to interact with OS. The program requests a service from a kernel that’s in the OS. System call is the only entry point to the kernel system. Few things that system calls can handle are memory, file process, protection, and networking. When system call is called, the program is interrupted and transfers the control to OS. The

biggest difference between system call and function call is that while system call requests a service from a kernel, a function call request is made by a program to perform a specific task.

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?

Cite: https://en.wikipedia.org/wiki/Glibc
https://stackoverflow.com/questions/13179102/how-do-the-standard-c-library-and-system-calls-work-together

glibc is the GNU C Library, which is the implementation of the C standard library. It provides core libraries such as Linux and kernel. Such libraries provide APIs which is heavily used in programming. The functionality it provides is different from the system calls because it is faster and does not usually require permissions since it's running along with the process. System calls, since they run in the kernel, needs to have access to everything in the system. Therefore, it requires more steps to call them than calling a library call.

The library interacts with OS system calls by providing programmers to access kernel related functions. It also provides syscall definitions.

PART 2

1. Explain in words how your implementation works

a. What were your assumptions? How did you interpret the specification?

First, my program will be able to open and read the input file I created. To apply doubly linked list in my code, I will need several node with pointers to store previous, current, and next element of each node. I will create loops to traverse the linked list and each loop will either add, remove, or print the elements. Every time it prints, it will print out the state of the list to the output file. If the user doesn't add anything and still orders to print, then it will print "EMPTY". Depending if the user adds head or tail, the order of the state will change but the output will show each nodes and how it's linked in between. For instance, if the

final result should be a,bb,cd then the output should show that a's next key is bb and bb's previous key is a while bb's next key is cd while cd's previous key is bb and since cd doesn't have a next key, it will show as TAIL instead. I will also implement functions for deleting certain keys. Based on what key the user wants to delete, the code will track where the key is located and delete the founded key(s).

b. What state do you need to store?

I will need to store nodes with pointers that has previous (head), current (key), and next (tail). There will be a variable of a new node to store and update the key, a variable for previous node, and a variable for last node.

The variable HEAD and TAIL will be replaced and stored with a new key, based on the given order "addtail or addhead". If the current key doesn't have any previous or next key, then it will point to NULL and print out either HEAD or TAIL instead.

c. What do your data structures look like?

Depending on where the new node is inserted or deleted, there will be different functions of doubly linked list. For instance, there will be functions for insertion at the beginning and insertion at the end. For deletion, there will also be functions for deletion of the first node, deletion of the inner node, and deletion of the last node. Within those functions, there will be loops to traverse the list back and forth as well as to print the current list if the instruction says to print.

d. What are the pieces of your code and how do they work together?

Once I create a struct Node function with node pointers of next, previous, and head, I will create a function to check if the list is empty. If list is empty and it's printed, it will print "EMPTY". Then, I will start off with another function with insertion at head. The next function with be insertion at tail. Following from that will be functions for deletion. There will also be a function to print out whatever we currently have in the list. The main function will call all these previous functions I made so it can run the program.

e. How does your algorithm work?

The doubly linked list in my code will have multiple links(pointers) that connects each node back and forth. The head doesn't have a previous node so it will just print out as "HEAD" and the tail doesn't have a next node so the next node will print out as "TAIL". Every nodes will have a current (key), previous pointer, and next pointer.

With the functions I have mentioned before, based on the given input, the program will either addtail, addhead, delete, or print.

1. initialize struct nodes

2. create a base case of isEmpty() to check if it's NULL.

3. method for insert at head

-create a node

-if there was nothing before, make this as the last node.

-else, update head

4. method for insert at tail

-create a node

-link previous last node to this new node

-since there is no next node, point it to NULL(will print out TAIL).

5. method for delete

-traverse the list to see if the string matches

-if the node with the searched string is found, re-link the previous node to the next node so it can skip this node.

6. method for print

-prints the current state of the list.

7. main function that calls all the previous functions

f. How do you plan to test your code?

Whenever I finish with each function, I will print out to check if it works properly. For instance, once I'm done writing a function for insertion at the beginning, I will test and print if addtail instruction is working by seeing the output. So I'll test by printing out the code whenever I'm done with a function.

g. How do you plan to build your code?
I plan to first write down functions for insertions then test it out to see if it gives correct outputs. Once that works, I will write functions for deletions and test it again.

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).

```
 GNU nano 2.9.3                                                sample.input

print
addhead "You"
print
addtail "Are"
print
addtail "Beautiful"
print
addhead "Wow"
addtail "Wow"
print
del "Wow"
print
del "Are"
print
del "You"
print
```

```
 GNU nano 2.9.3                                              expected.output

EMPTY
(You,HEAD,TAIL)
(You,HEAD,Are), (Are,You,TAIL)
(You,HEAD,Are), (Are,You,Beautiful), (Beautiful,Are,TAIL)
(Wow,HEAD,You), (You,Wow,Are), (Are,You,Beautiful), (Beautiful,Are,Wow), (Wow,Beautiful,TAIL)
(You,HEAD,Are), (Are,You,Beautiful), (Beautiful,Are,TAIL)
(You,HEAD,Beautiful), (Beautiful,You,TAIL)
(Beautiful,HEAD,TAIL)
```

3. How does Linux load a program from the file system and turn it to process? What are all the steps between?
Programs are stored on the disk and when we run it, the code gets its own section of memory and process is created. In detail, the Linux creates a content on disk then loads it into a memory and figures out

where the stack is going to start and then process it. When process runs, the code and static data gets loaded into memory and heap is given a location in the memory. Each process has its own address space and creates their own stack and heap on top of the data and instructions. The result of the process will have dynamic instance of code and data.

4. What is the exec function? How does it work and why do we normally use fork and wait with it?

Cite: https://devconnected.com/understanding-processes-on-linux/

An exec function replaces the current process image with a new program. When it calls a program, it provides it with some arguments. Once it passes to the function, it hits exec and OS loads the program up then it transfers control. The exec functions only returns if an error has occurred. We normally use fork because fork is a clone operation of parent and child. It takes the current process (parent) and clones it in a new process(child) with a new process ID. The execution of the cloned process will also start at the same instruction as the parent process. Basically, the child is identical to its parent except its context is unique. It has new stack, heap, registers, etc. The fork system call uses wait so that the parent executes last. After the fork, the child calls sleep which makes a process suspend for a set amount of time (1 second). Meanwhile, the parent issues a wait system call, which makes the process wait until the state of the child changes. Once the child exits, then the parent exits.

5. How does a C program in Linux get its arguments from the command line?

Cite: https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/

The C program in Linux get its arguments from the command line by being passed to the main() method with two arguments typically. First argument is the number of command line arguments and the second is list of command-line arguments. For instance,

int main(int argc, char *argv[]) {}

argc means argument count while argv means argument vector.

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?

Cite: https://www.programiz.com/c-programming/c-file-input-output

http://boron.physics.metu.edu.tr/ozdogan/OperatingSystems/week11/node4.html

In C, we use the functions fprintf() to write and fscanf() to read a file. In case of binary files, fread() and fwrite() are used.

To read a file from OS, we use a system call that specifies the name of the file and where the next block of the file should be put. The system keeps a read pointer to the location in the file where the next read is to take place.

To write a file from OS, we make a system call specifying both the name of the file and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer is updated whenever a write occurs.

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?

Cite: https://www.computerhope.com/jargon/s/special-file.htm

https://www.easytechjunkie.com/what-is-a-special-file.htm

Special files are a type of files stored in a file system. It exposes the device as a file in the file system and provides a universal interface for hardware devices so that tools for file I/O can access the device. There are two types of special files-character or block which indicates the type of device access provided. The special files have a special name that distinguishes it from other files. Special files can give commands to a device driver through I/O system calls. This makes it easier for the file to control a specific device or part of the computer system. We can use them with the instruction /dev/

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?

Cite: https://www.guru99.com/c-pointers.html

The pointers in C are variables that stores address of another variable. A pointer can also be used to refer to another pointer function. Without the pointers, we won't be able to save memory space and achieve faster execution time. We also won't be able to store address of another variable. Pointers are heavily used in linked list to point to the previous or next numbers correctly. We have used pointers in assembly to achieve singly linked list.

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?

Cite: https://www.guru99.com/stack-vs-heap.html

A stack is a temporary storage memory that stores temporary variables created by a function. Variables are declared, stored and initialized during runtime. When the computing is complete, the memory of the variable is erased. Stack helps us use Last In First Out method.

A heap is a memory to store global variables. All global variables are store in heap memory space. It supports Dynamic memory allocation. In C, variables are allocated and freed using malloc() and free().

Stack and heap are both memory areas allocated from the OS. Some key differences of stack and heap are stack is a linear data structure while heap is a hierarchical data structure. Stack memory never gets fragmented, but Heap memory can. Stack accesses local variables only while Heap can access variables globally.