Operating Systems
QUIZ 1

Nishaant Rastogi | 2020091

Q1.

If the proper header files (math.h) are included with the given code snippet there will be no compilation error while compiling the snippet using:

```
gcc -c  filename.c
```

Where filename is the file where we have our code snippet with appropriate header files. The above command creates an object file for the given code snippet with the name filename.o. This works because here we are not performing linking of the c file, we are just creating an object file. Creating an executable of the code snippet is not possible without a main function and would throw up errors.
Now in order to compile it along with a full-fledged C program say main.c, we will use the command:

```
gcc -o filename main.c filename.o -lm
```

This will create an executable file with the name filename which can be executed using the ./filename command. This works because now the code snippet has is linked to a C file with a defined main method, so now an executable file can be created without throwing up any errors.

One logical error I could find is the error of precision while adding the round values of 2 input decimals, see for example if we add 44.5 and 45.5 we will get 91(ASCII of [) instead of the desired 90(ASCII value of Z). This might lead to some unforeseen errors during runtime when the snippet is linked with a full-fledged program. In order to correct this error, it would be better to round the sum and then typecast it to int.

```
return (int)(round(a + b));
```

One more logical error can be the return type of the function, since we are adding floats the function is expected to return the sum in int data type but it is returning ASCII code of the int calculated.

Q2.

1)

Corrected Code:

```
extern printf
section .data
   format: db "%llx",0
section .text
   global main
main:
   push rbp
   mov rax, 0x1234567812345678
   xor ax, 0x11
   mov rsi, rax
   mov rdi, format
   call printf
   xor rax, 0x11
   mov rsi, rax
   mov rdi, format
   call printf
   pop rbp
   ret
```

Output:

```
12345678123456691
```

Reason:

We first move the hex value 1234567812345678 into the register rax. Then take xor of the value present in register ax (which corresponds to the last 16 bits of the rax register) with 0x11 and store it back into the last 16 bits of rax register. The last 16 bits hold the value 5678, xor it with 11 gives 5669, thus now the value in rax is 1234567812345669.

Next, we move these last 16 bits from ax to rdi. rdi is also a 64-bit register so these values are moved in the last 16 bits. rdi stores the first argument for syscalls. When printf is called it prints **1234567812345669**, and then the return value is stored in rax. printf returns the number of characters which is 16 in this case so 16 is stored in rax which would be 0x0000000000000010 in hexadecimal representation.

Next time when we xor it with 0x11 we get 0x0000000000000001 = **1** which is printed using the printf function.

2)

Output :

```
4294967294 4294967263 4294967261-2 -33 -35
```

Reason:

In this case we first declare an int variable x = -2, then an unsigned int y = -33 and then an int z = x + y.
When we first use the printf command using the unsigned int formatter (**%u**), then we get the output :

```
4294967294 4294967263 4294967261
```

This can be explained by the fact that a signed int like x is stored as 32bits in the memory and the negative integers are stored in the 2’s complement form. So for x = -2 the form was 1111111111111111111111111110, now when this no is formatted to unsigned int it does not take into consideration the signed bit and converts this binary to decimal directly which is 4294967294. Hence the printed value is just the conversion of signed int to unsigned int.
Similarly for y = -33, the signed binary form is 111111111111111111111011111 and the unsigned decimal form is 4294967263,
and for z = -35, the signed binary form is 1111111111111111111111011101 and the unsigned decimal form is 4294967261.
Now when the second time printf is called with all x, y and z formatted as int (**%d**), then the output is :

```
-2 -33 -35
```

These are simply the values assigned to the variables x,y and z and since these are in the range of int that is -2147483648 to 2147483647 there is no change in the values of the variables and the values printed.

Q3.

Output:

```
exit
before fork()
```

Reason :

When we run the program, we can see that a new shell is opened because first of all none of the print statements have been executed and when we exit the terminal the first print statement “before fork()” is printed on the terminal instead of killing the terminal.

From the above behaviour, we can conclude that after fork() is called the parent process is first put on wait using waitpid and the child process calls the execl system call to /usr/bin/bash which basically replaces the current running program with the program running in execl which is bash. Now the new terminal opens up as the child process get replaced with the execl program, and thus, the printf “done launching the shell”, of the original child process is never printed.

The new terminal is the program that replaces the child process, hence the rest of the commands to be executed in the child process do not execute. And when we exit from the shell we terminate the child process, hence we return back to the parent process and are able to see the output as “before fork()”. This statement was not printed before opening the new terminal because the string didn’t have a “\n”, this happens because the string “before fork()” is technically still not finished as its buffer was not cleared with any “\n”, hence instead of waiting for this process to finish, the compiler opens the child process (new bash terminal), now since the child process cannot be opened in the same line as the “before fork()” statement, because the print statement is part of output console of the original terminal and when we exit from the new terminal, the string “before fork()” is printed in the original terminal.

Q4.

SCHED FIFO is a scheduling algorithm in which tasks run to completion non preemptively, ie whichever process arrives first in the runqueue has to be handled or executed first and it is only after it has terminated chance is given to the other processes in the runqueue.
While under SCHED RR tasks run until they exhaust a fixed time quanta and are preempted to the ready queue if the task is not completed in the fixed quanta, and the unfinished task is later on again called on into the runqueue.
SCHED NORMAL is the scheduling class for normal processes and thus also called CFS or Completely fair scheduling. Under CFS, each processor gets a specific runqueue and each runqueue is a red-black tree. The processes with a smaller nice value (higher priority) receive a lower vruntime, increasing their fraction of the processor, while process’s with a larger nice value (lower priority) receive a larger vruntime, decreasing their fraction of the processor. This algorithm also maintains fairness by allotting specific time to all processes, if there are n processes then each receives 1/n processor time, the scheduler also maintains a minimum amount of time so that if no. of programs increase too much then swapping should not occur frequently as swapping also wastes time. The tree's internal nodes represent tasks or task groups, and these nodes are indexed by their vruntime values so that the internal nodes to the left have lower vruntime values than the ones to the right: In other words, tasks with the lowest runtime are present in the left subtree while those with relatively high vruntimes are in the right subtree. A preempted task would also go into the right subtree, thus giving other tasks a chance to move left in the tree. A task with the smallest vruntime has to be in the tree's leftmost node, which is actually the front of the runqueue.

For SCHED_RR the vruntimes are updated based on time quantum while for SCHED_FIFO it's based on the priority assigned to the process. Here the vruntime is calculated by the kernel in the sched_fair.c:__update_curr() program. It constantly updates the current and virtual time the process has spent on the processor and hence the selection of processes becomes increasingly dependent on a combination of factors including the weight load and nice values.

Q5.

1)

In this question we first initialise two character arrays (Strings) and take input into arr1 and call the copy_arr function which first copies the first 8 bytes (size of character pointer) of arr1 memory block to arr2 memory block, and then again copies “ABCD” to the first 4 bytes of arr2 memory block, using the memcpy function which is defined in the string.h library. The first parameter of memcpy is the pointer or reference object to which we have to copy into, the second parameter is the pointer which we have to copy from and the third parameter is the size of the memory block that needs to be copied from the second parameter to the first parameter. After returning from the copy_arr function, the arr2 is printed to STDOUT.

So output by example:

Input : abcdefghijklmnopqrstuvwxyz
Output: ABCDefgh

2)

If supposedly the address of a is equal to 0x1000 then the output according to the given snippet of code is:

0x10040x10010x1001

Now when we take b as an int * pointer it reads 4 bytes at a time. Therefore when we do b+1 it will move to the next 4 bytes and print that address. Therefore the address 4 bytes after address of a will be printed, i.e 0x1004.

Now when we typecast b to char * pointer it will read only 1 byte with every increment to find the next instance of char. Therefore when we print (char *) b+1, it will print the memory address of the next byte after the address of a which will be 0x1001.

The void *pointer reads the minimum bytes at each increment which is 1; Now when we typecast b to void * pointer it reads only 1 byte with every increment. Therefore when we print (void *) b+1, it will print the memory address of the next byte after the address of a which will be: 0x1001.