Quiz 1

## Q1.

The command used

gcc problem1.c -lm

./a.out

Warnings Encountered

→ overflow in conversion from ‘int’ to ‘char’

This error is Encountered due to converting a int of size more than 1 byte to a char which cannot have the size of more than 1 byte. This happens as we try to cast an integer of a larger size into a char, which can be resolved by taking a 1 byte integer to type cast into char.

→ Logical Error

when we will cast an integer to a char, the compiler will think of the integer as the ASCII value of the particular char. If we try to do this with large numbers that is of size more than that of 1 byte our result will be lost and will be of no use. In the conversion if try to print the char the ASCII value associated with the 1 byte integer will get printed. That byte being the 1st byte or the left most byte.

## Q2.

1. the register rax stores the hex value = 0x1234567812345678

after xored with 0X11. Now ax will have the hex value = 0x11. Therefore, the value printf will print is 17.

the xor of rax with value 0X11 will give = 0X1234567812345669

hence, 1311768465173141097 will be printed

2. the printf calls here at the first instant treats the 4 byte signed integer like an unsigned int. i.e.

-2 = 1111 1111 1111 1111 1111 1111 1111 1110(2’s compliment form)

will be treated like an unsigned which is 4294967294 as it will not consider the signed part.

Similarly,

-33 = 1111 1111 1111 1111 1111 1111 1101 1111

which is 4294967263 which is stored in y and hence

z = 4294967261 which is treated as a signed integer and therefore

z is equivalent to -35

however in the first print call we print each number as an unsigned integer therefore we get the given output of 4294967294 4294967263 4294967261

and when we treat each number as a signed int in the second print we get -2 -33 -35

## Q3.

In this code the printf(“before fork()”) doesn’t contain a new line charectar which means it will not print unless a new charectar show up or the end of the programme as printf is not buffered().

Now we fork() the process and inside the child process when pid==0 we start execl which will overwrite the whole child process with the desired process of the execl function.

i.e. the printf() below execl gets overwritten completely. Hence when we execute our programme. The child process runs as the parent process waits because of the waitpid() function call.

Hence A bash shell gets created as a child process. Which if we exit we get back to the parent process as the waitpid() lets the function to be executed and we see

the after `exit`

`before fork()`

## Q4.

SCHED\_FIFO is a real time scheduling policy that is a kernel component which decides which runnable thread will be executed by the CPU next. On a computer, we can run only a single task once, therefore in a multitasking computer the concept of “virtual runtime” was introduced. The vruntime of a task specifies when will the next part of the task be executed on an ideal multi-tasking CPU. SCHED\_FIFO uses FIFO(First in First Out) scheduling algorithm for scheduling, which means the process that requests the CPU first, gets executed first. In This scheduling algorithm when a process enters the queue, it first gets executed fully and after its execution the next process is executed therefore we don’t need to update the vruntime of such a program.

IN SCHED\_RR or Round Robin Scheduling → The round robin scheduling is similar to FIFO scheduling except a time slice is assigned to each process or a small amount of time is allocated to each process, which introduces the concept of vruntime and requirement of setting and updating it. It also has a process Queue where the processes wait for CPU to execute them.

In SCHED\_NORMAL we use a credit based policy in which the processes gets executed according to their priority or credit associated to them instead of First come First serve basis.

## Q5.

1. In this problem we call copy sizeof(p1) which is a char pointer i.e. has the size of 8 bytes. Hence it is the 8 bytes from the pointer provided as src which in this case is p1 which will be copied.

Let p1 be “abcdefghijkl”.

Then when we copy this to p2 with the given code

we get p2 = “abcdefgh”

After the second memCpy call the first 4 bytes again which were pointed by p2 will be replaced by “ABCD”.

Hence we will get our final output as “ABCDefgh”.

2.

*printf* ("%p\n", b);

it will print the address of a as b is assigned to the address of a.

0X1000

*printf* ("%p\n", b + 1);

It will print the address of b+1 which was an integer pointer hence 1 added to it should move by 4 bytes hence pointing after the current integer

0X1004

*printf* ("%p\n", (char \*)b + 1);

As b is typecasted to a char pointer it must assume the size of the address it is pointing to be of 1 byte and hence it will only move by 1 byte when 1 is added to it.

0X1001

*printf* ("%p\n", (void \*)b + 1);

As b is now a void pointer. Pointer arithematics is considered illegal on void pointers, and the output will vary from compiler to compiler on how they deal with such a problem for eg- my gcc compiler will give the output of

0X1001

which might be because of the way it treats void pointer arithematics