QUIZ 1

PROBLEM 1:

The snippet was compiled using - gcc -c QuizQues1.c

(QuizQues1.c was the name of the C program in which the snippet was present)

When the given code snippet is compiled as it is i.e. without making any changes, gcc gives a warning that either we provide a definition for round() function or include the math.h.

But if we compile after adding the math.h header as specified later that we have to assume that all header files are assumed to be present, then we do not get any warning or error and the code compiles successfully.

Now we make it a full fledged program by including the main() function and link it. The program now looks like:

```
#include<stdio.h>
#include<math.h>

char add(float a,float b)
{
   return (int)(round (a)+round (b)) ;
}

int main(){
   printf("%c\n", add(45.3, 22.7));
   printf("%d\n", add(45.3, 22.7));
   return 0;
}
```

This is how this program is compiled, linked and run along with the output:

```
[kern@artixcse231 Quiz 1]$ gcc QuizQues1.c -c
[kern@artixcse231 Quiz 1]$ gcc QuizQues1.o -lm
[kern@artixcse231 Quiz 1]$ ./a.out
D
68
```

-lm flag is added to incorporate the math.h header file.

The output can be explained as follows:

> 45.3 is rounded to 45 and 22.7 to 23. They are added and result in 68. 68 is the ASCII of 'D' so the output.
> Also it is considered that integer and char data types are many times interconvertible as the corresponding int of a char represents its ASCII.

Now the above program giving no compilation errors or expected output does not mean that there are no logical errors.

Whenever we convert a float value to int, there may be a loss of data as float can store values upto 3.40282e+38 while int stores value upto 2,147,483,647. Also since we are computing the rounded int value of added numbers but returning char type values, this is again a logical error. Doing this may result in dangerous data loss as int variable can store 4 bytes while char can only store 1 byte of information. So although int and char maybe seen as interconvertible with similar usage, type casting int to char results in a huge loss of information in case of large in values.

PROBLEM 2:

Part 1:

Initially, the program gives an error due to the incorrect call to the printf function
In my opinion, the question wants us to store 0x1234567812345678 in rax, 'xor' the lower 16 bits of this value with 0x11, and then print the lower 16 bits. This task is accomplished as:

```
SECTION .data
  str: db "%x", 0xA    ;0xA is the newline character

SECTION .text
  global main
  extern printf
main:
  mov rax, 0x1234567812345678
  xor ax, 0x11
  mov rdi, str      ;first argument of print
  mov rsi, 0        ;second argument of print first initialise to zero
  mov si, ax        ;here stored with what to print
  mov al, 0
  call printf
```

This part gives the output as:

```
5669
```

Explanation: rax being a 64-bit register would store only first 8 hexadecimal characters i.e. 0x12345678. Now if we 'xor' it digit by digit with 0x11, the lower 2 digits change to 69 as 8 xor 1 = 9(hexadecimal xor) and 7 xor 1 = 6. And other digits xored with 0 give the same digit.

In the last 3 lines of the program, as per my interpretation, it wants us to again xor the value in rax with 0x11 and then print using printf. The value stored in rax initially after the first call to printf would be 5 i.e. the return value of printf (i.e. the number of characters printed, 4 characters for the lower 16 bits in hexadecimal and one newline character). The code for that will be modified as:

```
  mov rdi, str
  xor rax, 0x11
  mov rsi, 0        ;to flush the value in rsi
  mov rsi, rax      ;move rax values to be printed
  mov al, 0
  call printf
```

The ouput for this part would be: 14

Explanation: value of rax, i.e. 0x5 is 'xor' with 0x11 hexadecimal digitwise. 1 xor 5 = 4 and 1 xored with 0 gives 1. Hence the output

Final complete code:

```
SECTION .data
  str: db "%x", 0xA    ;0xA is the newline character

SECTION .text
  global main
  extern printf
main:
  mov rax, 0x1234567812345678
  xor ax, 0x11
  mov rdi, str      ;first argument of print
  mov rsi, 0        ;second argument of print first initialise to zero to flush the register
  mov si, ax        ;here stored with what to print
  mov al, 0
  call printf
  mov rdi, str
  xor rax, 0x11
  mov rsi, 0        ;to flush the value in rsi
  mov rsi, rax      ;move rax values to be printed
  mov al, 0
  call printf
```

Part 2:

The output of the program is:

4294967294 4294967263 4294967261-2 -33 -35

This can be explained as follows:
In the first printf() statement, we try to print three numbers, x,y&z as unsigned int(%u). x is a normal integer stored at some memory location. When printf tries to access it as an unsigned int, it prints the following result: (maximum unsigned int + x + 1). This can be explained by the fact that we can imagine a round of numbers from zero upto 4294967295. printf start from 0 and adds moves the pointer in correspondance to the number it wishes to print. In this case, x=-2 so the pointer moves left from 0 to 4294967294. Or (4294967296+number)mod4294967295.
For y, we try to store a negative integer in unsigned int. What the compiler does is it assigns to the empty space in memory the value -33 and stores it there without a strict checking as to what that number is. We can also interpret it as storing the value (4294967296+(y=-33))mod4294967295 = 4294967263. So when we access it using %u printf, it outputs that. Similar to the case of x is the case of z which would compute 4294967263 + (-2) to give the answer as 4294967261 which might store it as -35 as the max possible value in int variable is 2147483647.

In the second print statement, we try to access the same variables as signed ints. So the output for the x and z part is pretty simple. Now since we tried to store -33 in unsigned int, it got stored as 4294967263. So since the signed int cannot acces a number larger than 2147483647, it would interpret it as 33rd number to the left of 0. Hence the output.

The 2 printfs in the above 2 parts of the queestion are quite different as in the first part the return value of printf i.e. the no of characters it prints is an important factor in assembly code and the printf is called from the assembly code by declaring it as an extern type function in the SECTION .text field. In the second part, a normal usage of printf under stdio.h header file is observed in the C code. In this part the return value of printf does not hold any significance but rather how it represents various data types (how it type casts). In the first part we print the hex values using %x as stored in the registers while in second part, we print the unsigned and int int values as %d and %u as stored in the variables.

PROBLEM 3:

The expected output would be:

```
before fork()
```

And then a new bash terminal would open.

But what we observe is that the current expected line that was expected to be printed is not buffered and the new terminal replaces and overwrites what would have been initially printed.

If in the printf() line that is expected to print "before fork()" is changed as:

```
printf("before fork()\n");
```

Then the line before fork() would be printed and then the control will move over to the new bash that opens up.
This happens because presence of buffer creates an indicatoin that when the content is to be pushed on the screen/output file.

The line: "done launching the shell" is not printed as after the call to execl(), the new process, defined in argument of the function begins to get executed by the processor and we loose the initial process that was running.

PROBLEM 4:

SCHED_FIFO scheduling uses First Come First Serve scheduling algorithm. In this the first process to enter the ready state is first executed completely and then the next process is allowed execution. So in this case the runtime would just be the total time for which it executes until termination or until it is forced to go to the waiting state as it is a non-preemptive method. We can say that this runtime only is the vruntime as it would not be required to come back in the ready queue. Also seeing from another perspective that the vruntime determines the process which has run for the least amount of time and needs to be sent for execution. In SCHED_FIFO, there can be 2 cases:

One that the process is terminated, in that case we can assign vruntime as infinity as that process would not go back to the ready queue.

Second, that the process is sent to the waiting state. In this case, it can be assigned vruntime as: vruntime of process at last of queue+1 and then after waiting state should be added to the end of queue(in accordance with First Come First Serve Policy).

Moreover if a process uses SCHED_FIFO, then it becomes a soft realtime process i.e. with high priority and negative nice value. So that it is executed before SCHED_NORMAL queue. It is different from the SCHED_NORMAL in the sense that SCHED_NORMAL uses a red black tree and a struct sched_entity to store and manipulate vruntime and uses a strict formula vruntime = runtime*nice_number

In the case of SCHED_RR, each process has the same priority and gets equal quanta of time for execution. So in this case whenever a process goes for execution, its runtime increase by the fixed quanta of time for which it executes. We can say that the vruntime also increases by the same amount as the time quanta.
In both SCHED_NORMAL and SCHED_RR, the vruntime is updated each time they are passed for execution. Unlike SCHED_FIFO, a process is not always completely executed at once it gets the processor. In SCHED_FIFO, there is no point of updating the vruntime as the process is executed until it is terminated i.e. its execution in the processor is finished so we can set it to infinity or as described above. So there wont be any point of updating the vruntime unlike in SCHED_RR and SCHED_FIFO.

PROBLEM 5:

Part 1:

In order to run this program, we need to add t header files: string.h (for memcpy) and stdio.h (for printf and scanf)

The output of the program will be as follows(when the input is abcde fghijkl):
Enter a string: abcde fghijkl

ABCDe fg

EXPLANATION:

1. There is a warning that the size parameter we are passing is based not on char but on char*. This warning occurs because the char pointer has a fixed size of 8 bytes and we wish to copy a string whose size may vary.

2. Clearly, the program asks for our input of a string using scanf and it accepts white spaces because of the modification %[^\n]s which means that the program will read until user enters a newline character.

3. Then the 2 char arrays, arr1 and arr2 are passed as arguments to the copy_arr() function. The first parameter is received as a character pointer. The size of a char pointer in C is 8 bytes.

4. 'memcpy()' has the following syntax:

```
void *memcpy(void *dest, const void * src, size_t n)
```

It copies 'n' number of bytes from the location 'dest' to the location 'src'.

5. On the first call of memcpy(), 8 bytes are copied from p1 to p2 as the sizeof(p1) returns 8. So now p2 contains first 8 characters that we entered as input. On the second call of 'memcpy()', the first 4 bytes are copied as "ABCD". Since memcpy() does not check for '\0' or overloading it does not alter the 8 bytes that were earlier set by our input string. It just overwrites the new 4 bytes “ABCD” at that location’s 1st 4 bytes.

6. Since there is call by reference whenever an arraay is passed as parameter, the change in p2 is reflected onto arr2.

7. When it is printed using printf, we observe the above mentioned output.

Part 2:

Again we have to add stdio.h header file for printf

If the value address of a is 0x1000.
Then the output would be:
0x10040x10010x1001

if we add a '\n' after each value being printed(just to see the output clearly), the output would be:

0x1004
0x1001
0x1001

Explanation:

'b' is declared as a pointer to an integer variable 'a' i.e. it points to the memory address at which 'a' is stored.

If we wish to increment the pointer by 1 (in the first case), then rather than pointing going to the next byte i.e. 0x1001, it points to the next int block as the size of int in C is 4 bytes. This means that (b+1) refers to the memory location next to the block at which 'a' is stored and 'a' would need 4 bytes. Hence 0x1004.

Now, in the second print statement, the pointer type is type casted to char* so it will be pointing toward as the same memory assuming that a char is stored at that location. So since the size of char is 1 byte, (char*)b+1 would point to the location (next byte) just next to the location where char would occupy 1 byte. Hence 0x1001.

Similarly when it is type casted to void* type, it points to the next byte address, assuming the current address would store a void variable. The size of the data type void is also 1, hence (void*)b+1 makes it point to the next byte i.e. 0x1001, next to where, current block of void is being pointed.

References:

https://www.tutorialspoint.com/cprogramming/c_data_types.htm
https://www.geeksforgeeks.org/chrt-command-in-linux-with-examples/
http://soundsoftware.ac.uk/c-pitfall-unsigned.html
https://stackoverflow.com/questions/7152759/what-happens-when-i-assign-a-negative-value-to-an-unsigned-int