Problem (1)

- **The first part**, the compilation of this code wouldn't give any errors or warnings.
- Command to compile without linking:
  >> gcc {filename.c} -c {filename.o}

- **In the second part**, assuming all necessary header files along with main function are included.
- Command to compile:
  >> gcc -o {filename} {filename.c} -lm
  >> ./{filename}
  (-lm flag to link math library for round function)
- **First logical erro**r: Assuming the user wants to add the two floats after rounding them to the nearest int and then return their sum, the function must have int return type, not the char.
- **Second logical error**: Return type char limits the range of the output to only -128 to 127 or 0 to 256 (since there are 8 bits). If the number is not in this range then it will use modulo operation to bring it in this range. So we might get an incorrect value for the sum.

Problem (2)

(1)

a) Corrected Code + Explanation

```
extern printf

mov rax, 0x1234567812345678      // move num to rax
mov rsi, rax                     // 2nd arg of printf (num to be printed)
xor rax, rax                     // as printf is varargs (variable no. of args)
mov rdi, format                  // 1st argument of printf (format)
call printf                      // printf(format, num)

mov rsi, rax                     // 2nd arg of printf (num to be printed)
xor rax, rax                     // as printf is varargs (variable no. of args)
mov rdi, format                  // 1st argument of printf (format)
call printf                      // printf(format, num)

format:
        db  "%20ld", 10, 0       // print 20 chars of num+“\n” (otherwise print
                                 empty space if number has less than 20 chars)
```

b) Output Explanation

```
>>  1311768465173141112
>>                    21
```

- Decimal form of number 0x1234567812345678 is printed
- Then return value of the first printf() call stored in “rax” is printed (Notice that the printf() returns the number of character that are successfully printed in this case these are 21 = len(num)+len(“\n”) = 20 + 1)

(2)

a) Output

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

b) Output Explanation

- size of int and unsigned int = 4 bytes
- x is stored as 11111111111111111111111111111110
- y is stored as 11111111111111111111111011111
- z = x + y is stored as 11111111111111111111111011101

- When we use %u to print x,y,z then the number stored is interpreted as an unsigned integer so the most significant bit is treated as positive so we get output as shown in the first line
- When we use %d to print x,y,z then the number stored is interpreted as a signed integer so the most significant bit is treated as negative so we get output as shown in the second line

Differences in the output of (1) and (2)

- In (1) we give the hexadecimal representation of the number and then simply print it in decimal form using assembly language
- In (2) we give the decimal representation to both unsigned and signed variables and then print them in decimal treating them first as unsigned (using %u) and then as a signed integer (using %d) using C language

Problem (3)

a) Output

>>

(No output to stdout)

b) Output Explanation

- We are simply using fork system call and then making parent process wait till the child process in not completed.
- In the child process we are using execl system call to execute the bash program.
- FORMAT: int execl(const char *path, const char *arg, …, NULL);
- String which is called before forking the process "before fork()" is not printed as we have not provided “\n” to it.
- So the buffer doesn’t flushes the output to the stdout at that moment and save it for later.
- Meanwhile, child process executes the new bash program so nothing is seen on the stdout as it is the child process bash.
- One way to exit from the bash will be to kill or close the terminal.

Problem (4)

- In SCHED_FIFO, FIFO scheduling, the process that comes first is executed without any preemption.
- Vruntime in this case accumulates for the running process as long as it is running and keeps updating depending on its priority as a decay factor.
- After the termination of the process, the control is passed to another process.

- While, in SCHED_RR, Round Robin scheduling, the process runs for a particular time slice and then preempts and goes to the end of the list of processes of its priority.
- In SCHED_NORMAL, Completely Fair scheduling, the normal process (non-real time) the processes have static priority.
- So virtual runtime (or vruntime) is the same as normal runtime and is independent of priority as contrasted with FIFO and RR.

1. *vruntime = runtime * (nice_number + 21)*
2. *vruntime is unsigned long*
3. *Implementation:*
    a. *AVL Tree*
    b. *Red black tree*

Problem (5)

(1)

a) Output

>> Enter a string:hellobrolol
>>ABCDobro

b) Output Explanation

- Declare two char arrays of size 100 each and then take a string input in one of the char array say arr1.
- Use %[^\n]s to take the complete line as input from the stdin to arr1.
- Then call the copy_arr function which calls the memcpy library function and copies the first sizeof(char*) bytes from arr1 to the arr2.
- Note that the sizeof(char*) is 8 bytes so only first 8 bytes i.e. “hellobro” is copied
- Then we again call memcpy function and copy the 4 bytes “ABCD” to the first 4 bytes of arr2.
- So the arr2 now looks like “ABCDobro”.
- Finally, print the arr2 to the stdout.

(2)

a) Output

>> 0x1004 0x1001 0x1001

b) Output Explanation

- a is an int variable with a value 2. Assume address of a is 0x1000.
- b is an int pointer variable pointing to a. So b stores 0x1000.
- Any addition or subtraction on a pointer p, changes its value
  from p to p + val*(size of the variable pointed by p)
- So b+1 = 0x1000 + 1*sizeof(int) = 0x1000 + 0x4 = **0x1004** (size of int is 4 bytes)
- Similarly we can typecast the pointer to any other type consequently its size referred also changes and hence output of addition and subtraction on the pointer also changes.
- So (char*) b+1 = 0x1000 + 1*sizeof(char) = 0x1000 + 0x1 = **0x1001** (size of char is 1 byte)
- Similarly (void*) b+1 = 0x1000 + 1*sizeof(void) = 0x1000 + 0x1 = **0x1001** (size of void is 1 byte)