Name - Sagar Suman

Roll No. 2019197

OS Quiz -1

Ans1 -
a) If we consider the code snippet given in the question, then on compilation we get warning regarding implicit declaration of function 'round'. We get this warning because we have not declared the
function 'round' before it. And as we have not included the math.h header file in the code, we didn't have any built-in function definition that matches with the declaration of the function 'round'.
One way to resolve this warning is to include math.h header file.
If we consider, we have included the math.h header file in the code then the compilation of the code will not throw any error or warning. Because in this case math.h will provide the built-in function definition
that matches with the function round.
If we save the code in file with name 'q1.c', then command to compile the c file is -

```
    gcc -c q1.c
```

in which we are using -c switch to compile only.

b) If the code is compiled and linked with full fledged program, then in that case code will lead to logical error. Because if we consider the add function to add two floating numbers. Then the output we
will get is not the desired one. Instead we get the result as rounded off addition of two floating point numbers. So even if the program is correct in terms of syntax, it is logically giving wrong result.

Ans2 -
b) From the first printf statement, we get output as 4294967294 4294967263 4294967261 for the values of x, y and z respectively. And,
from the second statement, we get output as -2 -33 -35 for values of x, y and z. This is mainly because in the first statement we are printing unsigned integer values and in second case we are printing integer values.
In the code, we have a addition of x as int which is -2 and y as unsigned int which is -33. During the addition, x's values is taken as unsigned int which is 4294967294 (2^32 - 2). And then we add it to y which is -33. So z becomes 4294967294 -33 = 4294967261.
As z is of type int, unsigned addition result gets converted back to integer, which is -35.During printing, we are using arguments for x,y and z as u, which is an unsigned int. So thats why we get output for x,y(2^32 - 33) and z in unsigned format. In second time, we passed the arguments as d, which is int.
The values of x,y and z are taken as int and finally printed.

Ans3-
output of the code - before fork() done launching the shell.
Explanation - In the code we are firstly, printing 'before fork()'. Then we are calling fork() system call, which creates child along with the parent process. In case of parent process, we are using waitpid system call
to wait for its child to get terminated. In case of child process, we encounter execl system call. We should not have printed 'done launching the shell' if execl system call was terminated successfully. Execl system
call only returns when it encounters the error, which in this case is happening. Reason being we are going inside /usr directory, with execl. For this reason, execl is returning -1. Also,instead of /usr/bin/bash in the file argument
of execl, if we had /bin/bash then our output could be only 'before fork()', i.e execl would not have returned.

Ans4 -
In SCHED_FIFO we use the First In-First Out scheduling algorithm. This scheduling method is Non-preemptive. It implements just one queue which holds the tasks in the order they come in. On the other hand, SCHED_RR Uses Round Robin scheduling algorithm. It is an algorithm used for Preemptive scheduling. And SCHED_NORMAL: Uses Default Linux time-sharing scheduling algorithm or simply the standard round-robin time-sharing policy.

Ans5-
a) In this part, we are taking string as input and stored it in character array arr1. Then we are calling copy_arr function in which, we are using memcpy function to copy content of arr1 to another character array arr2.
Memcpy function in c, takes three argument - pointer to array where content should be copied, pointer to array to array from which we have to copy and number of bytes to be copied. But here we are only coping first 8 bytes of arr1, as the size of character pointer (in my machine - 64bit ,i.e 8) is passed as an argument for number of bytes to copy, in first memcpy function. In case, arr1 has length which
is less than 8, then whole arr1 is copied to arr2. After this, we are again calling memcpy function to copy 'ABCD' as the first four bytes in arr2. Now for this, length of arr2 (which is less than or equal to 8) can have two possibilities.
First is when arr2 have length greater than or equal to 4, so in this case memcpy simply copy 'ABCD' in first four bytes and leave other bytes untouched. Second is when arr2 have length less than 4, in this case we get an output of length 5, in which first four bytes are
'ABCD' and fifth element is garbage value. Based upon this we get output, when we -
- take input string of length>8: get output as string of length 8, into which first 4 characters are 'ABCD' and last four characters belong to original input.
- take input string of length<=8 and >=4: get output string of same length as input,with first 4 characters as 'ABCD' and rest characters (if there are any) are same.

- take input string of length < 4 : get output string of length 5, in which we first four characters are 'ABCD' and last character is garbage value.

b) In this part, we get the 3 memory locations as an output.
Explanation :- In the code snippet, we are creating the pointer variable, which points to memory location of integer variable a (which contains 2 as a value).
It will be easier to understand the output, if we consider this memory location as 0x1000.
In first printf statement, we are doing addition of integer pointer with one. So as per pointer arithmetics, we will get next integer location, which is 4 bytes (sizeof(int)) next to the current pointer address.
Which gives our first value as 0x1000 + 0x4 = 0x1004.
Similarly, in second printf statement we are type casting int pointer to char pointer and then adding 1 to it. So as per pointer arithmetics, we will get next char location, which is 1 bytes (sizeof(char)) next to this pointer address.
Which gives our second value as 0x1000 + 0x1 = 0x1001.
And lastly,we are type casting int pointer to void pointer and then adding 1 to it. Now void pointer actually points to some type, so incrementing the void pointer is actually the undefined behavior. Compiler in this case uses
takes the next location which 1 byte away from itself. Although it is advisable in void pointers to cast to appropriate type first and then do addition.
Hence, in this case we get 0x1000 + 0x1 = 0x1001.
Therefore as an output we get- 0x1004 0x1001 0x1001