Jatin Kumar Sharma
Roll No.: 2020563

Answer-1

Writeup:

Compilation of this code snippet doesn't result in any kind of errors or warnings. After compiling and linking the code snippet with a full fledged program produce logical error for bigger numbers, when program returns result of the operation, i.e. addition of the given numbers after rounding them to the nearest integer, it returned in memory of size of char (1 byte) as the return type of the code snippet is char. Therefore, any pair whose sum results exceed the size of a byte(sum not in range from 0 to 255) will not produce output as expected if the actual sum is in the range [0, 255] then it will store as it is simply but if the sum is greater than 255 then the remainder we get after dividing the actual sum will be stored (i.e truncated sum)

The output that is to be printed will also depend on how we access it, i.e. char or int
For char, simply the ASCII corresponding to the number stored will be printed, whereas if accessed as int(%d), then stored number in the range of [0, 127] will be printed as it is stored but any number greater than 127 will be printed in its 2's complement form of a signed 8 bit number, as int also contains negative numbers and in 1byte the range for such numbers is [-128,127].

Commands:

1. For compilation only: Consider the name of the file having code snippet is first.c
   gcc -c first.c -o first.o //compilation of first.c
2. After attaching the file to full fledged program in addition to the above:

```c
#include <stdio.h>

char add(float a,float b);

void main(){
   char c = add(122,200);
   printf("%c\n", c);
}
```

gcc -c main.c -o main.o // compiling main file(main file have definition of add)
gcc main.o first.o -o result -lm //for linking to main file

Answer-2: 1

Observation:

when executed the program the output of both of the print statements is as follows:
1311768465173141097
1311768465173141112

Explanation:

The value that was moved to 64bit register %rax initially is 0x1234567812345678, hence the value stored in 2-byte register %ax will be 0x5678. Now xor been taken of the value 0x11 and %ax, that means xor been taken of 0x5678 and 0x11 and (0x5678^0x11 = 0x5669) hence now the value stored in register %ax will be 0x5669 and subsequently the value stored in register %rax would be 0x1234567812345669. In subsequent steps, this value has been moved to 64-bit register %rsi so that it can be printed. The print uses %llx format to print these values so in the output the decimal value of 0x1234567812345669 that is 1311768465173141097.

Now again xor been take between %rax and 0x11, i.e. (0x1234567812345669 ^ 0x11) this value will be stored to %rax, Now the value after this operation in %rax is 0x1234567812345678 which is equal to the value stored in register %rax initially, the value that will be printed will be the decimal equivalent of 0x1234567812345678, hence the output will b 1311768465173141112.

The xor of %rax taken with 0x11 has been neutralized by taking xor again with the same number(0x11).

Answer-2: 2

```
int x=-2;
unsigned int y = -33;
int z;

z = x + y;
```

When a number is stored in memory, if it is positive then it is stored as it is, but if it is negative then it is stored in the form of its 2's complement form under the size allotted(i.e 2's of 32 bit number). Here, It does not matter whether we store a number as int or as unsigned int, if it's negative then it will be stored as its 2's complement, if in the range [0,4294967295] it will be stored as it is and if it is greater than 4294967295, the remainder we get after dividing that number will be stored as a result of truncation. As here

x = -2 in memory will be stored in its 2's complement as 4294967296-2 = 4294967294.
y = -33 in memory will be stored in its 2's complement as 4294967296-33 = 4294967263.
z = x+y = 4294967294 + 4294967263 = 8589934557 % 4294967296 = 4294967261

Now when we access them as unsigned integers then the numbers are accessed as it is (i.e x will be printed as 4294967294, y as 4294967263 and z as 4294967261 which is the output of the first print statement of the given program), but when we access then as integers then the most significant bit out 32 of is considered as sign bit and the no will b accessed in its 2's complement form as for the given program when accessed as int:

x is printed as 4294967294 - 4294967296 = -2
y is printed as 4294967263 - 4294967296 = -33
z is printed as 4294967261 - 4294967296 = -35

That is the output of the 2nd print statement of this program.

Jatin Kumar Sharma
Roll No.: 2020563

Answer-3

Observation:

when executed the program doesn't print anything until and unless we put a ' \n' character in the string.

After putting a \n it only prints <before fork()> in CLI and stops.

Explanation:

When the program is executed the very first printable instruction it encounters is to print before fork() which don't print as the output just doesn't get flushed to the screen without the \n which it demands, after putting /n at last of "before fork()", before fork() printed out.

Now the process is forked, and the parent process is exposed to wait till the child process finishes or terminate; the child process executes execl command with the first argument is the file to execute which is "/usr/bin/bash" here and 2nd argument "bash" and third argument or say the last argument is NULL. execl is a system call that transfers the process image or replaces the current process image with a new process image created by a call to the fork system or any other system like pthread. Here the new process image is to call the "bash" and launch a new shell. As soon as the execl command executes, a new process takes the place of the current process(the pid of the newly generated process remains the same as of the process that will be replaced). The shell launches successfully, and the program terminates. Now, as the child process get replaced by the new process that launch shell there will not be any existence of the child process containing the exec syscall and the print statement to display dialogue "done launching the shell", hence after execl call this print statement won't execute as there is no existence of this print statement after the system call.

The shell stores the recently used commands of a location; when we ran this fork program and after its termination, if we check for the history commands, then we won't be able to find those commands using the UPArrow key of the keyboard, which verifies that a new shell had been created.

Jatin Kumar Sharma
Roll No.: 2020563

Answer-4

A program that uses SCHED_FIFO scheduling runs till it doesn't get finished, whereas in SCHED_RR scheduling there is a quantized time for which a process runs, and if it doesn't finish in the meantime then it again queued to the last of all the process and wait for its turn to continue again from where it left and preempted after it finishes, whereas for SCHED_NORMAL scheduling the process which consumes a is selected from a red-black tree.

SCHED_FIFO (First In First Out) scheduling is known as FCFS(First Come First Serve). FIFO or FCFS works on the fact that whichever task arrives first in the run queue should be executed first. Until it doesn't get terminated; any other process of the run queue can not come to perform, the time which is consumed from starting to termination of the process is its vruntime which varies from process to process.

SCHED_NORMAL also known as CFS or Completely fair scheduling In this scheduling, each process is stored in a specific run queue which is kind of a red-black tree, in which the nodes of trees represents a task has to be executed and these processes are arranged in a fashion such that the node having lowest vruntime presents at left most node whereas the process having greatest runtime presents at the rightmost node.

A task with the smallest vruntime has to be in the tree's leftmost node, which is actually the front of the runqueue.

Answer-5: 1

‘

Output:

Whatever be the length of the input string, in the output the string is truncated to 8 always with having its first 4 characters replaced with ‘ABCD’.

Explanation:

The main function creates arr1 and arr2 and then stores a sequence of char (String) into it and calls for the copy_arr() function having arr1 and arr2 as its two arguments.

copy_arr() accepts its first argument as char pointer and 2nd as an array. This function first copies the content of p1 of size p1 itself to p2. Since p1 is a pointer and the size of a pointer is 8-byte always, it will first copy 8 bytes of p1 to p2; when the call is made to memcpy it will copy 4byte of ‘ABCD’ and replace the initial bytes of array p2. Now when the program returns to the main function it calls the printf and prints in stdout.

Here the size of the output string can not be greater than 8 as we have copied the 8 bytes of p1 to p2; also, we have again copied 4 bytes of ABCD to the first 4 bytes of p1. Hence it will always print ABCD along with the char that doesn’t overlap.

Example output:

enter a string: hello → ABCDo

Hereafter, the first memcpy command, p2, will have hello, then after the 2nd memcpy call, the first 4-byte of hello will be replaced with ABCD and prints ABCDo.

enter a string: 123456789 → ABCD5678

Hereafter the first memcpy command, p2 will have 12345678, then after the 2nd memcpy call, the first 4 bytes of hello will be replaced with ABCD and print ABCD.

Answer-5: 2

Output:

The output comes out as follows in the system compiler:
0x7fffe6502450 0x7fffe650244d 0x7fffe650244d

Explanation:

Any pointer is of size 64 bit or says 8 bytes, here a pointer is created which points to variable 'a' as we have provided the address of a given as 0x1000, then the pointer b will also have the address of a that is 0x1000, Now we have printed pointer (b+1), it will look for the location b+1 which would be 0x1000+0x04 = 0x1004 which will be printed if we print pointer b+1.

Again when we typecast the int pointer b to (char *) pointer or say (void*), it will again point to the location of **a** that is 0x1000; hence the value returned for (char *p) will be 0x1000.

Printf have been called as:

p r i n t f (' '%p ' ' , (ch a r *) b + 1) ;
p r i n t f (' '%p ' ' , (v oid *) b + 1) ;

That means the value of the pointer b is to be printed after adding 1 to it. So both of the last printf statements should print the location that 0x1000 + 1 = 0x1001

Hence the output sequence corresponding to the print statements considering the provided address of a will be:

0x1004 0x1001 0x1001