Operating System (CSE 231)

Quiz - 1

Name : Shivansh Choudhary
Roll No : 2020127

Solution 1 :

We have tried to run the code provided that stdio.h and math.h are already included in the code.

When we compiled the code the using :

```
gcc filename.c
./a.out
```

The following errors came up :

```
/usr/bin/ld: /tmp/ccLAjQAW.o: in function `add':
q1.c:(.text+0x26): undefined reference to `round'
/usr/bin/ld: q1.c:(.text+0x43): undefined reference to `round'
collect2: error: ld returned 1 exit status
```

The warning simply states that there was no linking of the code to the round() function in the math library or that the math library wasn't linked. The compilation is successful when we link the math library in C to the following code using the -lm flag to the gcc compiler.

Command we use to compile the code:

```
gcc  -lm filename.c
./a.out
```

Since in the snippet given it is clear that return type is char from the add function. In this case the given snippet will not have any logical error and if the value which is to be returned is greater than equal to 128 then the compiler will reverse the cycle and will start from -128 and in printing it will char which corresponds to the ASCII value.

But if we look at the mathematical definition of add then what add does is to add two numbers and return the result which is the addition of two operands. In this implementation again when the number overflows 127 it will revert back to -128 and produce the output accordingly. So to make the implementation of the addition function correct we will change the return data type from char to int.

Solution 2 :

a) Since at the starting of the code the rax register is storing `1234567812345678` in hexadecimal form.
Which when converted to binary is
`1001000110100010101100111100000010010001101000101011001111000`

In the next instruction when we are performing xor of `0x11(i.e 10001 in binary)`, with an rax register.
The output of the xor is getting stored in rax and then rasx is getting copied from rsi and the resultant answer will be printed on the screen which is the value stored in the rax register.

Output at this point will be
`1234567812345669`

Then when the printf returns it's control to main, it will push the number of printed characters i.e `16 in decimal` into rax. So now rax will store the value `16 in decimal i.e 10 in hexadecimal.`

Then again rax will be xor with given `0x11(i.e 10001 in binary)`

and the result will be stored in rax and in the next encountered printf the value of rax will be displayed onto the screen.
Output at this point will be
1

Hence the final output will be
12345678123456691

And the corrected code will be :

```
mov rax, 0x1234567812345678
xor rax, 0x11
mov rdi, rax
call printf
xor rax, 0x11
mov rdi, rax
call printf
```

The error is because we can't mov a smaller register into a big register.

b) The output of the code is

4294967294 4294967263 4294967261 -2 -33 -35

The first printf uses %u which denotes unsigned integer which typecast the given signed integer to unsigned integer.
If the value becomes negative then it performs the reverse cycle by adding the given signed integer to the max value of unsigned integer i.e 4294967295. Hence the output went as written above.
The second printf uses %d which denotes signed integer and it prints the values of x,y,z as it is.

Solution 3 :

Explanation:

The output is because of absence of newline character or flush the output stream due to which all the print buffer i.e "before fork()" gets stored in memory and the standard output (stdout) stream isline buffered, hence it will wait for the process to end or when it will encounter a newline character, it will print all that stored data into the stdout stream and hence onto the screen.

Then the code encounters the fork() system call. This creates two processes one is child and other is parent and both these processes will have the same print memory and both start the execution from the forking point.

We have used waitpid() system call to give instruction to parent to wait till the child process executes completely.

Since pid of child is equal to zero, the child process will enter the if condition and it will encounter the exec() system call and the current ongoing process is replaced by the instruction stated in exec() instruction.

The execl("/usr/bin/bash","bash",NULL); since this instruction is successful; it will not return and will replace the current ongoing process with the new bash process and the current ongoing child process will be overtaken and all buffer memory and process of child will be lost. Then , when the control returns to the parent process, it is completely executed; it will print the statement "before fork()" and then exit.

So on the basis of above explanations, the output of the program would be

before fork()

Solution 4 :

SCHED_NORMAL (CFS) :

The virtual runtime (vruntime) is primarily used to account for how long a process has run & then CFS will pick up the process with the smallest virtual runtime (vruntime). The vruntime is then calculated on the grounds of weight and nice value of the process.

If :

Nice value	Weight
0	1
>=0	Less than 1
<0	Greater than 1

When a nice value equals 0, the process will have spent vruntime equal to the real time and when it is greater than 0, hence to maintain balance between the processes, the real runtime will be less than vruntime and in the last case when nice value is less than 0, the vruntime value will get updated less frequently and vruntime will be less than real time.

Where vruntime=(time executed)*(weight factor)

SCHED_RR :

All processes in Round Robin scheduling have equal timeslice.
The vruntime of each process will be updated after a regular time interval and since every process will get an equal timeslice to execute, so vruntime will be updated with equal value every time.
PIT will interrupt when the process timeline will get over and it will be pushed back into a ready queue and it will again wait for his turn to come.

SCHED_FIFO:

SCHED_FIFO is a simple scheduling algorithm without time slicing. **SCHED_FIFO** can be used only with static priorities higher than 0, which means that when a **SCHED_FIFO** thread becomes runnable, it will always immediately preempt any currently running **SCHED_OTHER**, **SCHED_BATCH**, or **SCHED_IDLE** thread.
With the first arriving process having the minimum vruntime and the later arriving process having vruntime higher than it.

Solution 5 :

a) The size of the pointer in C is 8 bytes.
When we called the copy_arr() function with the given parameters the memcpy() function copies the first 8 bytes from p1 to p2 and then when the next memcpy() instruction executes the given string "ABCD" overwrites the first 4 bytes of the p2.
Then when printf is called it prints the 8 bytes of arr2 in which starting 4 characters will be "ABCD" and the rest depends on the length of the input string.

So if the:
Input is : iloveopsys
Output will be : ABCDeops

b) As the size of integer pointer in C is 4 bytes and the address of given integer variable a is given as 0x1000 , hence when we b=&a , b is also assigned the same address and then when we used %p which is used to printing the address of a pointer, then when we print b+1 then it will print (1000 + 4) = 1004. This is because when we are doing +1 to the fetched address (1000) it will add 4 to the current address as b is itself an integer data type and as stated int occupies 4 bytes of

memory space. So it will take the next 4 memory spaces and it's address value will increase by 4. Hence the output will be 0x1004.
Then in the next printf we encounter, (char *) b will give 1000 and then when we add 1 to it , the output will be 1001. This happens because the address of b is 1000 and b is type casted into char which occupies 1 byte of memory spaces. So adding 1 to the address space will occupy the just next memory address i.e 1001 . Hence the output will be 0x1001.
Then in the next printf we encounter, (void *) b will give 1000 and then when we add 1 to it , the output will be 1001. This happens because the address of b is 1000 and b is type casted into void which occupies 1 byte of memory spaces. So adding 1 to the address space will occupy the just next memory address i.e 1001 . Hence the output will be 0x1001.
Hence the output of the code will be :
0x10040x10010x1001