OS Quiz 1

Aayush Gakhar

2020006

Q1.

I put the code snippet in a file named q1.c.

Then ran it using 'gcc -c q1.c -o q1 -lm' command.

```
[art@artixcse231 quiz1]$ gcc -c q1.c -o q1 -lm
q1.c: In function 'add':
q1.c:5:14: warning: implicit declaration of function 'round' [-
Wimplicit-function-declaration]
    5 | return (int)(round (a)+round (b)) ;
      |              ^~~~~
q1.c:1:1: note: include '<math.h>' or provide a declaration of
'round'
  +++ |+#include <math.h>
    1 | // #include <math.h>
q1.c:5:14: warning: incompatible implicit declaration of built-in
function 'round' [-Wbuiltin-declaration-mismatch]
    5 | return (int)(round (a)+round (b)) ;
      |              ^~~~~
q1.c:5:14: note: include '<math.h>' or provide a declaration of
'round'
```

It returned 1 error. round() function was undefined. We need to include the math.h header file for it to work.

Then I included math.h.

Then ran it using 'gcc -c q1.c -o q1 -lm' command. -lm flag is for the math library.

It ran without any errors.

If we run it with a fully fledged program it will run without any program errors. There is a logical error that the return type of add func is char but it returns value int. This may cause a error in the program as the correct value would not be reflected by a char.

In the add function the floats are rounded before addition which may give wrong answer for the addition. for example add(1.5,4.5) return 7. For correct addition the rounding should be done after adding the numbers, if we desire integer output, which would give add(1.5,4.5) as 6.

If we need float output then we should not round the parameters and just return a+b.

Q2.

1.

output:
0x1234567812345669
0x2

First we move 0x1234567812345678 to rax. The we xor rax with 0x11 which makes rax = 0x1234567812345669. Now we use printf to print rax value. which is printed in first line.

Next we xor the value present in rax with 0x11. We print rax and get 0.

This happens because when we call printf, it stores the length of the printed string in rax register.
So value in rax becomes 0x11 = 17 which is length of "1234567812345669\n". Then we xor with 0x11, it becomes 0x0.

code:

```
extern printf

section .data
    format db "%llx", 10, 0

section .text
    global main

main:
    push rbp          ; this is done to align the stack
    mov rax, 0x1234567812345678
    xor rax, 0x11     ; here ax was giving opcode mismatch error
    mov rdi,format    ; printf needs the format as first parameter
    mov rsi, rax      ; parameter for printf
    call printf
    xor rax, 0x11
    mov rdi,format
    mov rsi, rax
    call printf
    pop rbp           ;pop rbp from stack
```

run using:

```
nasm -f elf64 q2.asm -o q2.o
gcc q2.o -o q2 -no-pie
./q2
```

givs:
1234567812345669
0

We see that rax gets changed by printf so the value by second prinf is different. This can be rectified if we stored rax before printf and then restored it back. This has been done in the following code.

```
extern printf

section .data
    format db "%llx", 10, 0

section .text
    global main

main:
    mov rax, 0x1234567812345678
    xor rax, 0x11
    mov rdi,format
    mov rsi, rax
    push rax          ; saves rax to stack
    call printf
    pop rax           ;restore rax from stack
    xor rax, 0x11
    mov rdi,format
    mov rsi, rax
    push rax
    call printf
    pop rax
```

This gives output:
1234567812345669
1234567812345678

line 1: 0x1234567812345678 ^ 0x11 = 0x1234567812345669
line 2: 0x1234567812345669 ^ 0x11 = 0x1234567812345678

2.

output:
4294967294 4294967263 4294967261 -2 -33 -35

explaination:

negative integers are stored in 2's complements notation.

-k is stored as (2^n - k) where n is the no of bits available. So the available numbers are -2^{n-1} to $2^{n-1}-1$.

%u prints the no as unsigned int so it just prints to value of 2's complement without converting it to negative,

here 2^{32}=4294967296.
So,
-2 = 4294967296-2 = 4294967294
-33 = 4294967296-33 = 4294967263
-35 = 4294967296-35 = 4294967261
%d is the integer format. it prints the correct values of all numbers. so we get -2 -33 -35.

Q3.

```c
int main()
{
    pid_t pid1;
    printf("before fork()\n");
    if((pid1=fork())>0)
    {
        waitpid(pid1,NULL,0);
    }
    else if(pid1==0){
        execl("/usr/bin/bash","bash",NULL);
        printf("done launching the shell\n");
        exit(0);
    }
    else{
        perror("fork()");
    }
}
```

output:
before fork()
done launching the shell

explain:
main() prints "before fork()".

Then pid1=fork() creates a forked process. In parent pid1=child process pid. In child pid1=0.

So for print if condition is true. waitpid(pid1,NULL,0) makes parent process wait for child process to exit.

Meanwhile child process executes the else part of condition as pid1==0.
It calls execl() which executes a separate file given to it. It is given the path to bash shell which is "/usr/bin/bash". So it executes the bash shell. Then it prints "done launching the shell".

The last else is if pid1<0 which is the case when fork process encounters an error.

Q4.

SCHED_NORMAL is the name for default method for CFS or Completely fair scheduling (CFS). Under this each process is assigned a vruntime. The process are stored in a red black tree and the process with least vruntime is given higher priority. vruntime is the runtime of the process till then multiplied by their niceness factor. It uses a virtual clock to calculate how much time a process would be given in a ideal completely fair processor.

SCHED_FIFO stands for first in first out. Under this a process is run till completion. The process which comes first will be run completely before other processes get a chance. This is a non-preemptive process. Vruntime does not make sense as the process is completed in a single go. If we want to calculate we would put it as 0 so that out program has the highest priority. The priority is given by when the task is initialized and when it gets a chance to run it runs to completion.

SCHED_RR is round robin in which each process is given a fixed amount of time and in a cyclic manner. Each process would be run for equal amount of time slice. The vruntime should be and calculated such that these tasks ar arranged in the tree in a circular fashion.

Q5.

1.

```c
#include <stdio.h>
#include <string.h>

void copyarr(char* p1,char p2[]){
    memcpy(p2, p1, sizeof(p1));
    memcpy(p2,"ABCD",4);
}
int main(){
    char arr1[100];
    char arr2[100];
    printf("Enter a string:");
    scanf("%s",arr1);
    copyarr(arr1,arr2);
    printf("%s",arr2);
    return 0;
}
```

The program creates 2 char arr: arr1 and arr2. Then it takes a string from user and stores it into arr1. Now it calls a function copy_arr with arr1 and arr2 as parameter.

In copy_arr() memcpy function is used.
memcpy(void* dest, void* src, size_t size_of_src)

p1::arr1 p2::arr2
memcpy (p2,p1,sizeof(p1)); : copies p1 to p2 the first sizeof(p1)=8 elements. so max length of p2 is 8.

memcpy (p2,"ABCD",4); : copies "ABCD" to the first 4 indexes of p2.

Then return back to main.

main then prints string stored in arr2.

The first 4 characters are "ABCD" as they were copied to arr2.
The max possible length of arr2 is 8. the next 4 characters are the 4-7 index characters from arr1.

so,
input length<=4 ==> output = "ABCD"
input length>4 ==> output = "ABCD"+(4-7 index characters from input)+sometimes garbage values as string is printed till it comes across '\0' character.

eg:

Enter a string:asdfghjk
ABCDghjkw

w is garbage value.

Enter a string:qwerty
ABCDty

here no garbage as the '\0' character comes from the input.

2.

```c
#include <stdio.h>

int main(void){
    int a=2,*b=NULL;
    b=(int*)0x1000;
    printf("%p\n",b+1);
    printf("%p\n",(char*)b+1);
    printf("%p",(void*)b+1);
    printf("%lu",sizeof(void));
}
```

The output:

0x1004   // this is b+1
0x1001   // this is (char*)b +1
0x1001   // this is (void*)b +1

address of a==0x1000

line 1:

sizeof(int)==4.
As size of int is 4 bytes the address of consecutive int are 4 bytes apart. b points to int a. b+1 points to the next int after a which will be at 4 bytes after address of a. So it is at 0x1004.

line 2:

sizeof(char)==1.
As size of char is 1 byte the address of consecutive char are 1 byte apart. b points to address of a. b+1 points to the next char after a which will be at 1 byte after address of a. So it is at 0x1001.

line 2:

sizeof(void)==1.
As size of void is 1 byte the address of consecutive void are 1 byte apart. b points to address of a. b+1 points to the next void after a which will be at 1 byte after address of a. So it is at 0x1001.