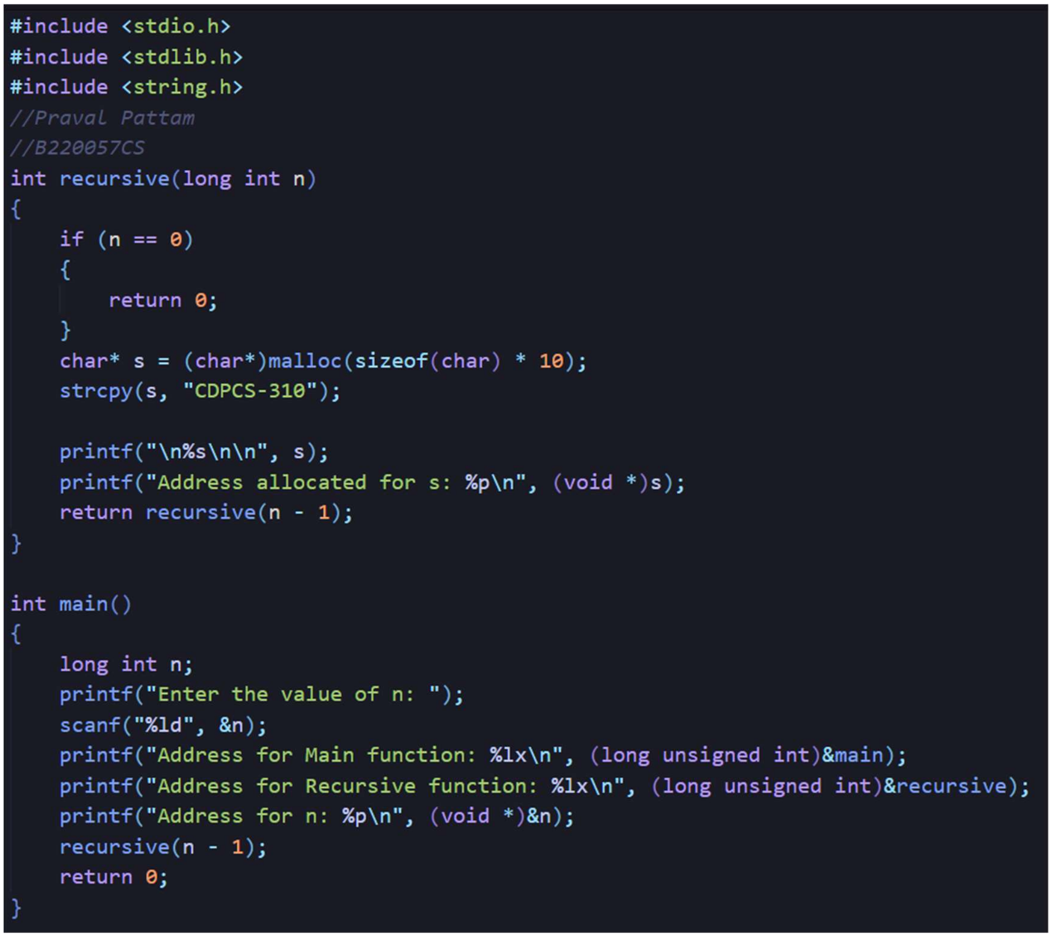
OS ASSIGNMENT 1A

Praval Pattam
B220057CS
CS04

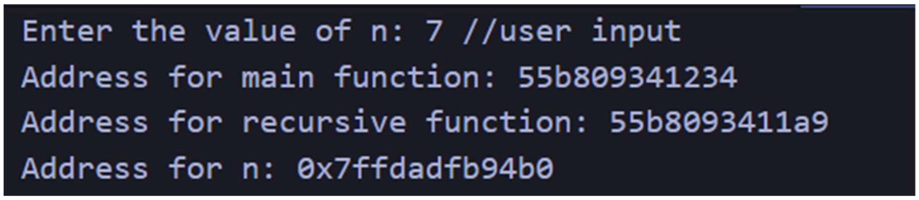
Q1)

A) Code



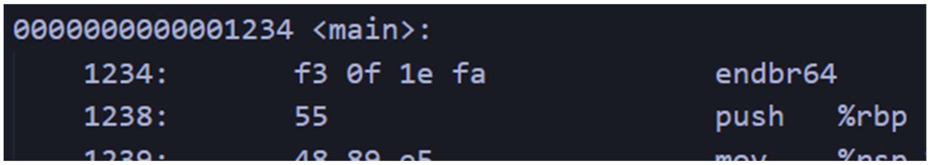
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//Praval Pattam
//B220057CS
int recursive(long int n)
{
    if (n == 0)
    {
        return 0;
    }
    char* s = (char*)malloc(sizeof(char) * 10);
    strcpy(s, "CDPCS-310");

    printf("\n%s\n\n", s);
    printf("Address allocated for s: %p\n", (void *)s);
    return recursive(n - 1);
}

int main()
{
    long int n;
    printf("Enter the value of n: ");
    scanf("%ld", &n);
    printf("Address for Main function: %lx\n", (long unsigned int)&main);
    printf("Address for Recursive function: %lx\n", (long unsigned int)&recursive);
    printf("Address for n: %p\n", (void *)&n);
    recursive(n - 1);
    return 0;
}
```

1) On running the program for user input n = 7
We get the following outputs before entering the recursive function.



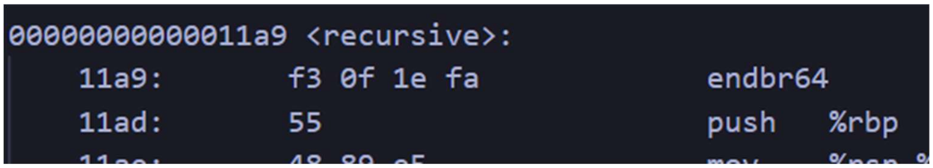
```
Enter the value of n: 7 //user input
Address for main function: 55b809341234
Address for recursive function: 55b8093411a9
Address for n: 0x7ffdadfb94b0
```

Upon running the command "objdump -sRrd a.out" after the execution of a.out
We can see that in the output we have the offsets addresses which is the same as the last 4 digits in main and recursive function addresses printed above
for the main function



```
0000000000001234 <main>:
    1234:       f3 0f 1e fa             endbr64
    1238:       55                      push   %rbp
    1239:       48 89 e5                mov    %rsp
```

And the recursive function



```
00000000000011a9 <recursive>:
    11a9:       f3 0f 1e fa             endbr64
    11ad:       55                      push   %rbp
    11ae:       48 89 e5                mov    %rsp,%
```

2) Stack Growth Direction:

The stack typically grows downwards, meaning each new frame is allocated at a lower memory address than the previous one.

3) Stack Frame Size:

The stack frame size for each recursive call includes the space for the local variables and the return address. This can vary depending on the system and compiler, but typically it includes the size of the long int parameter and the return address.

We can estimate this by checking the stack frame addresses in the debugger and subtracting the addresses of the frame 1 and frame 0:

0x7fffffffeaa0 - 0x7fffffffead0 = 0x30

0x30 = 48 in decimal system

So size of each frame is 48 x 64 bits since we are using a 64-bit architecture.

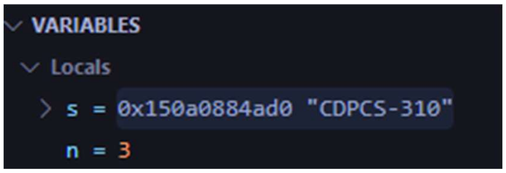
4) Heap Location and Growth Direction:

The heap is located in a different page of memory than the stack and typically grows upwards, meaning each new allocation is at a higher memory address than the previous one.

5) Contiguity of malloc()'ed Memory Areas:
The memory areas allocated by malloc are not guaranteed to be contiguous. There can be extra space between their addresses due to the way the heap manager handles memory allocation and fragmentation.

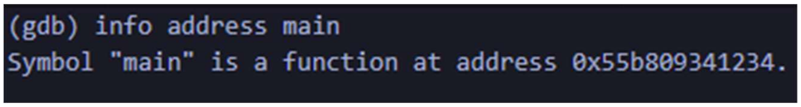
B)

1) While using the gdb debugger for an input value of n = 7
For which the value passed to the recursive function will be n – 1 i.e 6
So after n / 2 recursive calls i.e 3 recursive calls we will have the local variables



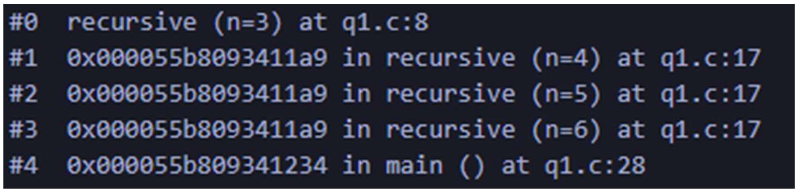
```
VARIABLES
  Locals
    > s = 0x150a0884ad0 "CDPCS-310"
      n = 3
```

On typing “info address main” we get the address of the main function.



```
(gdb) info address main
Symbol "main" is a function at address 0x55b809341234.
```

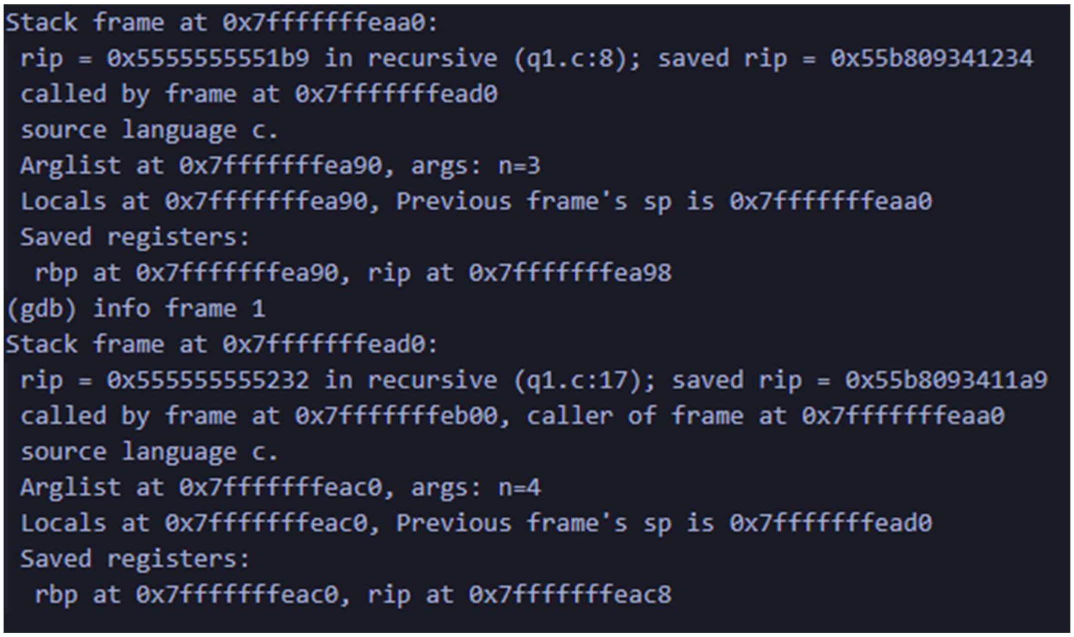
2) On typing “info stack” in the gdb we get



```
#0  recursive (n=3) at q1.c:8
#1  0x000055b8093411a9 in recursive (n=4) at q1.c:17
#2  0x000055b8093411a9 in recursive (n=5) at q1.c:17
#3  0x000055b8093411a9 in recursive (n=6) at q1.c:17
#4  0x000055b809341234 in main () at q1.c:28
```

Which shows us the caller stack frame that the program has at that point in time

3) On typing “info frame” in the gdb we get the follows



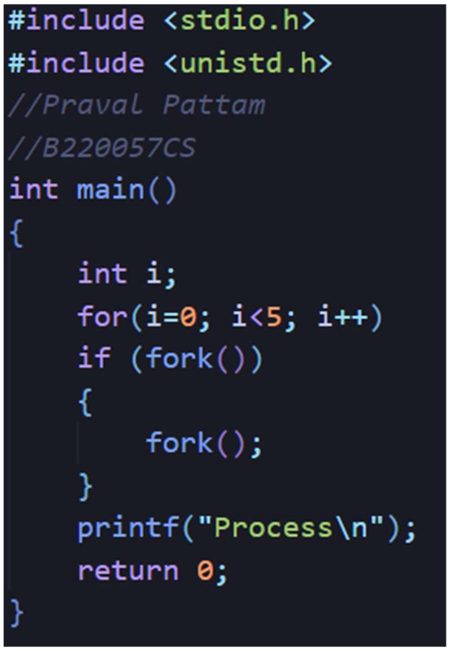
```
Stack frame at 0x7fffffffeaa0:
 rip = 0x5555555551b9 in recursive (q1.c:8); saved rip = 0x55b809341234
 called by frame at 0x7fffffffead0
 source language c.
 Arglist at 0x7fffffffea90, args: n=3
 Locals at 0x7fffffffea90, Previous frame's sp is 0x7fffffffeaa0
 Saved registers:
  rbp at 0x7fffffffea90, rip at 0x7fffffffea98
(gdb) info frame 1
Stack frame at 0x7fffffffead0:
 rip = 0x555555555232 in recursive (q1.c:17); saved rip = 0x55b8093411a9
 called by frame at 0x7fffffffeb00, caller of frame at 0x7fffffffeaa0
 source language c.
 Arglist at 0x7fffffffeac0, args: n=4
 Locals at 0x7fffffffeac0, Previous frame's sp is 0x7fffffffead0
 Saved registers:
  rbp at 0x7fffffffeac0, rip at 0x7fffffffeac8
```

We can look at each specific frame using “info frame <frame number>”
and it outputs

- the address of the frame.
- the address of the next frame down (called by this frame).
- the address of the next frame up (caller of this frame).
- the language in which the source code corresponding to this frame is written.
- the address of the frame’s arguments.
- the address of the frame’s local variables.
- the program counter saved in it (the address of execution in the caller frame).
- which registers were saved in the frame.

Q2)

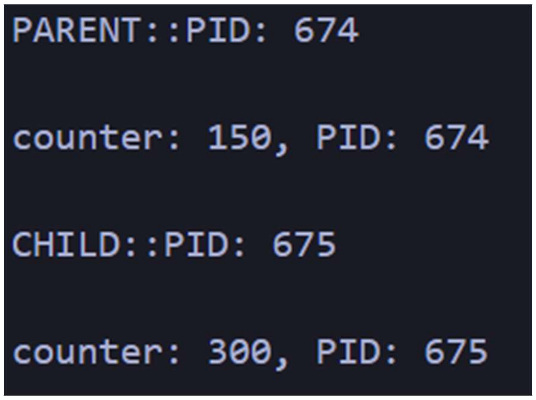
This creates 3 processes (including the parent process) each time. Since this is iterated over 5 times, a total of 3^5 – 1 = 243 - 1 child processes will be created, resulting in 243 processes overall as we have to include the original process.



```
#include <stdio.h>
#include <unistd.h>
//Praval Pattam
//B220057CS
int main()
{
    int i;
    for(i=0; i<5; i++)
    if (fork())
    {
        fork();
    }
    printf("Process\n");
    return 0;
}
```

Putting the print statement after the if statement as shown above will help us to find all the processes that are created. On running this code, we get the word “Process” printed 243 times.

Q3)

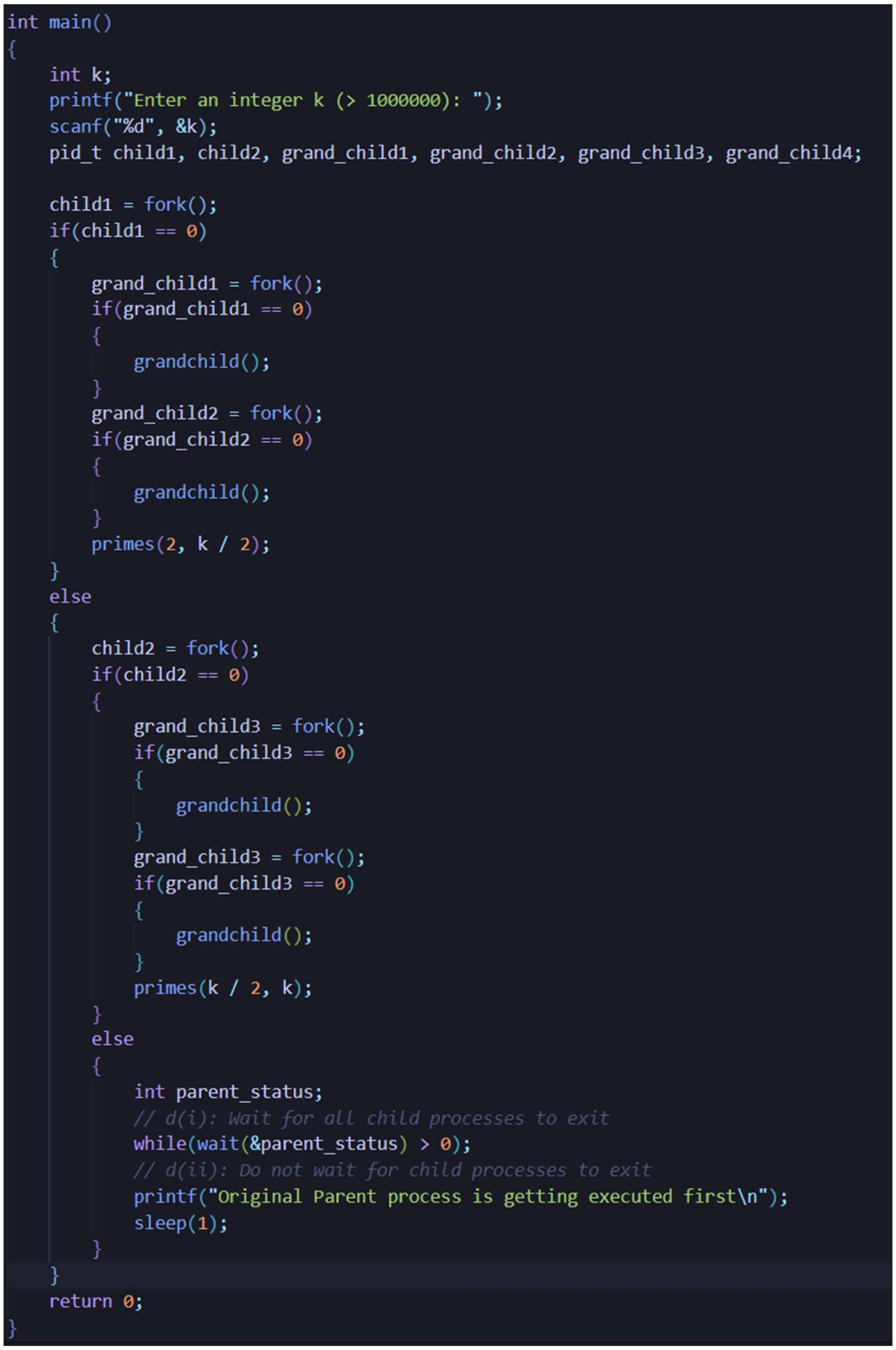


```
PARENT::PID: 674

counter: 150, PID: 674

CHILD::PID: 675

counter: 300, PID: 675
```

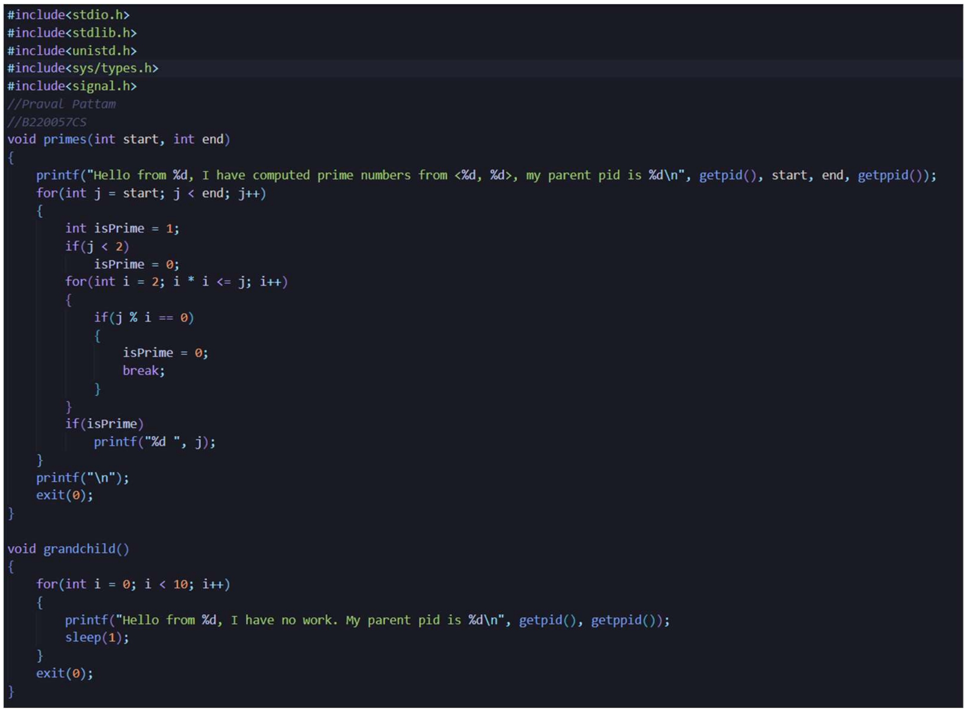
On running the given code (output give above), we can see that the counter value in the parent process and the counter value in the child process is 150 and 300 respectively and no, counter variable is not shared between the parent process and child process. as there is memory protection for all the variables used in each process. In this case calling fork() allocates a copy of the parent’s code to the child but the variables are not the same as they are localized to their respective processes.

Q4)

Code for the main function



```
int main()
{
    int k;
    printf("Enter an integer k (> 1000000): ");
    scanf("%d", &k);
    pid_t child1, child2, grand_child1, grand_child2, grand_child3, grand_child4;

    child1 = fork();
    if(child1 == 0)
    {
        grand_child1 = fork();
        if(grand_child1 == 0)
        {
            grandchild();
        }
        grand_child2 = fork();
        if(grand_child2 == 0)
        {
            grandchild();
        }
        primes(2, k / 2);
    }
    else
    {
        child2 = fork();
        if(child2 == 0)
        {
            grand_child3 = fork();
            if(grand_child3 == 0)
            {
                grandchild();
            }
            grand_child3 = fork();
            if(grand_child3 == 0)
            {
                grandchild();
            }
            primes(k / 2, k);
        }
        else
        {
            int parent_status;
            // d(i): Wait for all child processes to exit
            while(wait(&parent_status) > 0);
            // d(ii): Do not wait for child processes to exit
            printf("Original Parent process is getting executed first\n");
            sleep(1);
        }
    }
    return 0;
}
```



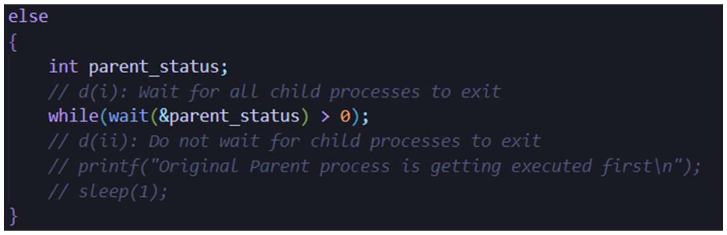
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<signal.h>
//Praval Pattam
//B220057CS
void primes(int start, int end)
{
    printf("Hello from %d, I have computed prime numbers from <%d, %d>, my parent pid is %d\n", getpid(), start, end, getppid());
    for(int j = start; j < end; j++)
    {
        int isPrime = 1;
        if(j < 2)
            isPrime = 0;
        for(int i = 2; i * i <= j; i++)
        {
            if(j % i == 0)
            {
                isPrime = 0;
                break;
            }
        }
        if(isPrime)
            printf("%d ", j);
    }
    printf("\n");
    exit(0);
}

void grandchild()
{
    for(int i = 0; i < 10; i++)
    {
        printf("Hello from %d, I have no work. My parent pid is %d\n", getpid(), getppid());
        sleep(1);
    }
    exit(0);
}
```

Header files and the remaining functions for printing primes and also grandchild()

Now

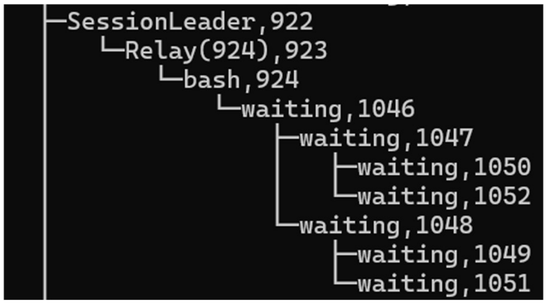
D)

i) Running the code where the parent process waits until all of the child processes are terminated done by commenting the print statement and the sleep(1) statement in the last else condition.



```c
else
{
    int parent_status;
    // d(i): Wait for all child processes to exit
    while(wait(&parent_status) > 0);
    // d(ii): Do not wait for child processes to exit
    // printf("Original Parent process is getting executed first\n");
    // sleep(1);
}
```

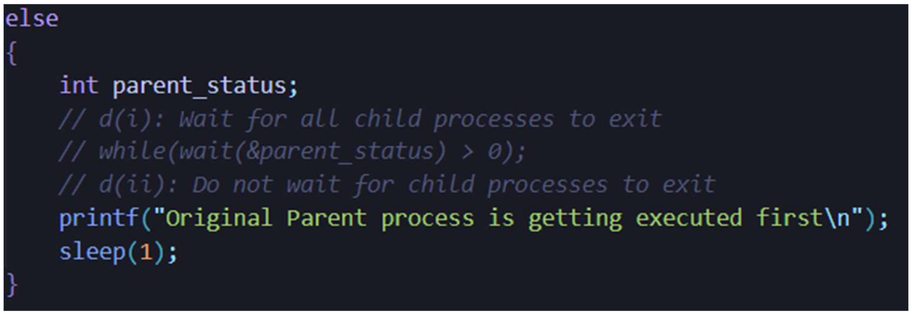
In this case, the parent waits for all child and grandchild processes to finish. The process hierarchy will remain intact until all processes exit when using the pstree command we get the following output.



```
│ ├─SessionLeader,922
│ │   └─Relay(924),923
│ │       └─bash,924
│ │           └─waiting,1046
│ │               ├─waiting,1047
│ │               │   ├─waiting,1050
│ │               │   └─waiting,1052
│ │               └─waiting,1048
│ │                   ├─waiting,1049
│ │                   └─waiting,1051
```

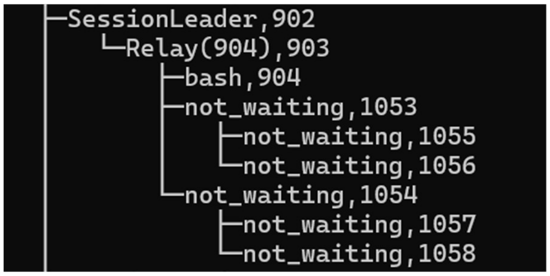
We can clearly see that the parent process (pid = 1046) that has called two child processes (pid = 1047 and pid = 1048) which give raise to four grand childern processes (pid = 1050, pid = 1052 and pid = 1049, pid = 1051 respectively).

ii) Running the code where the parent process terminates can be done by commenting the while(wait() > 0) line like shown below.



```
else
{
    int parent_status;
    // d(i): Wait for all child processes to exit
    // while(wait(&parent_status) > 0);
    // d(ii): Do not wait for child processes to exit
    printf("Original Parent process is getting executed first\n");
    sleep(1);
}
```

In this case the parent process (pid = 1052) exits before its child processes (pid = 1053 and pid = 1054) which give raise to four grand childern processes (pid = 1055, pid = 1056 and pid = 1057, pid = 1058 respectively) finish execution hence making the child processes orphans. This makes the init process(pid = 903) the new parent as shown in the process hierarchy below.



```
├─SessionLeader,902
│ └─Relay(904),903
│   ├─bash,904
│   ├─not_waiting,1053
│   │ ├─not_waiting,1055
│   │ └─not_waiting,1056
│   └─not_waiting,1054
│     ├─not_waiting,1057
│     └─not_waiting,1058
```