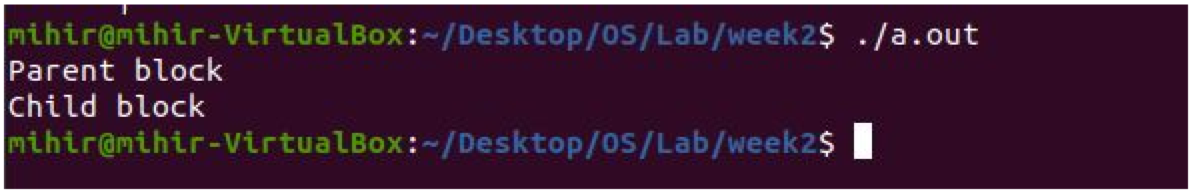
OS assignment 2

I. Test drive the various examples of fork that have been discussed so far in the class. Understand the output generated and in fact try editing those codes with possible combinations such as changing the parent / child definitions, point of forking, etc. this will help you to better appreciate the working of fork system calls.

Question 1.

```
#include <stdio.h>
int main()
{
   int pid;
   pid = fork();
   if (pid < 0)
         printf("Fork failed\n");
   else if (pid == 0)
         printf("Child block\n");
   else if (pid > 0)
         printf("Parent block\n");
   return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
Parent block
Child block
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$
```

Manual trace:

Output will be:
Parent block
Child block
(Note that the order can be different as it is dependent on the kernel)

Reasoning:

At forking point, main splits into two processes or in other words, gives rise to a new process C1 which is the child of main.

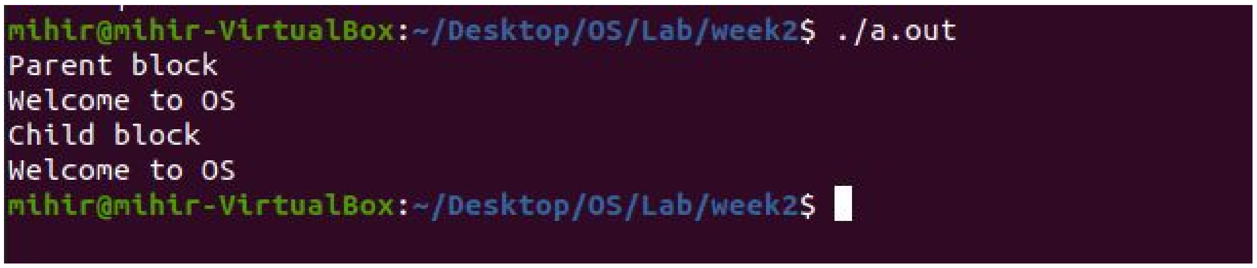
Fork returns 0 when the control is with the child process and a value >0 when the control is with the parent process.

So, when the control is with the parent process, pid is > 0 and **“Parent block”** is printed. Similarly when the control is with the child block, pid is = 0 and **“Child block”** is printed.

Question 2.

```
#include <stdio.h>
int main()
{
   int pid;
   pid = fork();
   if (pid < 0)
          printf("Fork failed\n");
   else if (pid == 0)
          printf("Child block\n");
   else if (pid > 0)
          printf("Parent block\n");
   printf("Welcome to OS\n");
   return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
Parent block
Welcome to OS
Child block
Welcome to OS
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$
```

Manual trace:

Output will be:

```
Parent block
Welcome to OS
Child block
Welcome to OS
```

(Note that the order can be different as it is dependent on the kernel)

Reasoning:

At forking point, main splits into two processes or in other words, gives rise to a new process C1 which is the child of main.

Fork returns 0 when the control is with the child process and a value >0 when the control is with the parent process.

So, the parent and the child processes execute their respective if blocks.

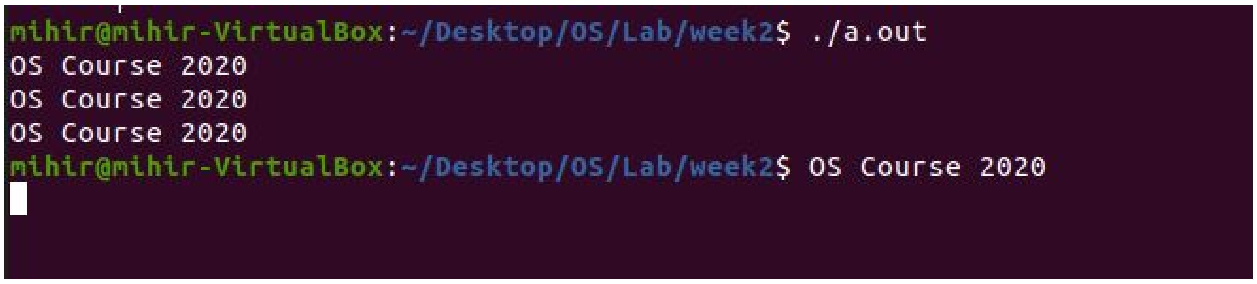
Now, when the parent process has finished executing its if block (pid > 0) it moves forward and encounters a printf statement (“Welcome to OS”). Since this printf statement is outside all the if blocks, hence the parent process executes it and **“Welcome to OS”** is printed.

Similarly, when the parent process has finished executing its if block (pid > 0) it moves forward and encounters the same printf statement (“Welcome to OS”) and **“Welcome to OS”** is printed again.

Question 3.

```
#include <stdio.h>
int main()
{
   fork();        //A
   fork();        //B
   printf("OS Course 2020\n");
   return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
OS Course 2020
OS Course 2020
OS Course 2020
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ OS Course 2020
```

Manual trace:

Output will be:
OS Course 2020
OS Course 2020
OS Course 2020
OS Course 2020

Reasoning:

On A, M splits into two processes, M and C1 (or gives rise to a new process C1), where C1 is the child of M. M then moves forward and on B again splits into two process M and C2, where C2 is the child of M. On moving forward, M encounters the printf statement and **"OS Course 2020"** is printed. **(1)**

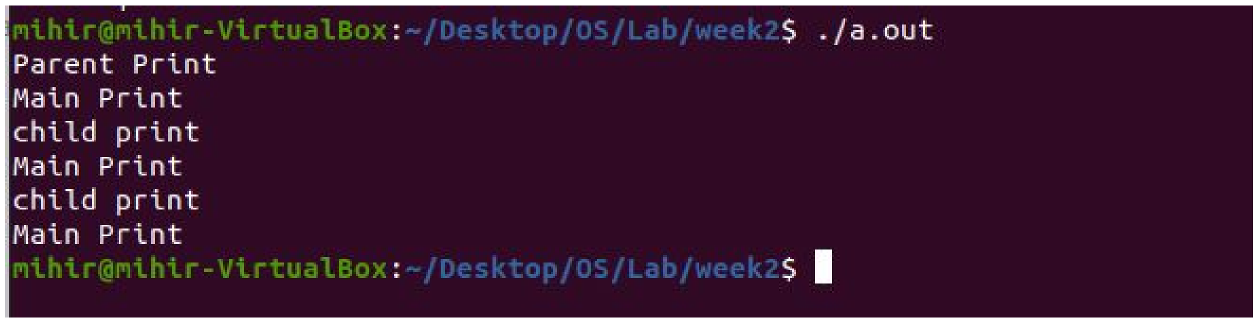
Now, for C1, program starts from B. On B, C1 splits into two processes C1 and C3 (or gives rise to a new process C3), where C3 is the child of C1. On moving forward, C1 encounters the printf statement and **"OS Course 2020"** is printed. **(2)**

For C2 and C3, since they are generated at B, the only line visible to them is the last printf statement and **"OS Course 2020"** is printed by each one of them. **(3) (4)**

Question 4.

```
#include <stdio.h>
int main()
{
int pid;
pid=fork();     //A
if (pid<0) fprintf(stderr,"failed fork \n");
else if (pid==0)
{
fork();     //B
printf("child print \n");
}
else if (pid>0)
printf("Parent Print \n");
printf("Main Print \n");
return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
Parent Print
Main Print
child print
Main Print
child print
Main Print
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$
```

Manual trace:

Output will be:
Parent Print
Main Print
child print
Main Print
child print
Main Print

Overall **6** print statements.

Reasoning:

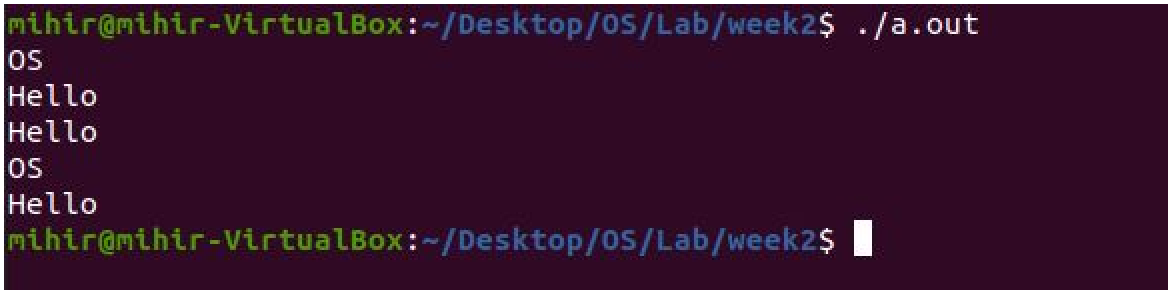
On A, M splits into two processes, M and C1, where C1 is the child of M. When the control is with the parent, pid > 0 so M will execute its if block and **“Parent Print”** will be printed. Since the last printf statement is outside any if block, it will be printed by all the created processes.

When the control is with the child process C1, pid becomes equal to 0. Now, C1 enters its if block and splits into two processes on encountering the fork statement, C1 and C2, C2 being the child of C1. both C1 and C2 will now print the immediate next printf statement and the last printf statement. So, the number of prints by C1 and C2 will be 2 each and 4 in total.

Question 5.

```
#include <stdio.h>
int main()
{
int pid;
pid=fork();     //A
if (pid > 0) {
fork();     //B
printf("OS \n");
}
printf("Hello \n");
return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
OS
Hello
Hello
OS
Hello
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$
```

Manual trace:

Output will be:
OS
Hello
Hello
OS
Hello

Overall **2** times **OS** and **3** times **Hello**.

Reasoning:

On A, M splits into two processes, M and C1, where C1 is the child of M. When the control is with the parent, pid > 0 so M will execute its if block and encounter another fork statement. Now, M again splits into two processes M and C2, C2 being the child of M. On moving forward M prints **“OS”** and then **“Hello”**.

C1 cannot access the if block so it directly prints **“Hello”**.

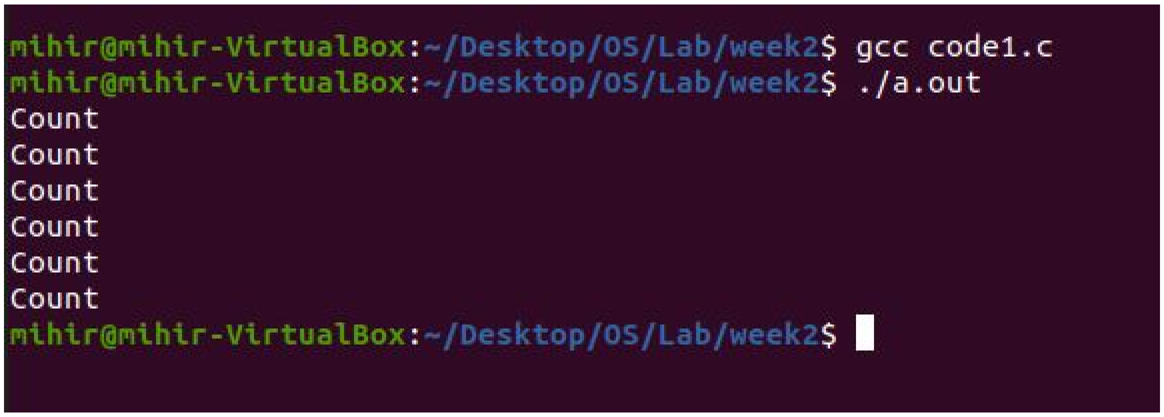
C2 prints **“OS”** and then **“Hello”**.

<u>II. Fork Practice Set</u>

Question 1.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
        pid_t pid;
        pid=fork();       //A
        if (pid!=0)
        fork();           //B
        fork();           //C
        printf("Count \n");
        return 0;
}
```

Output ss:



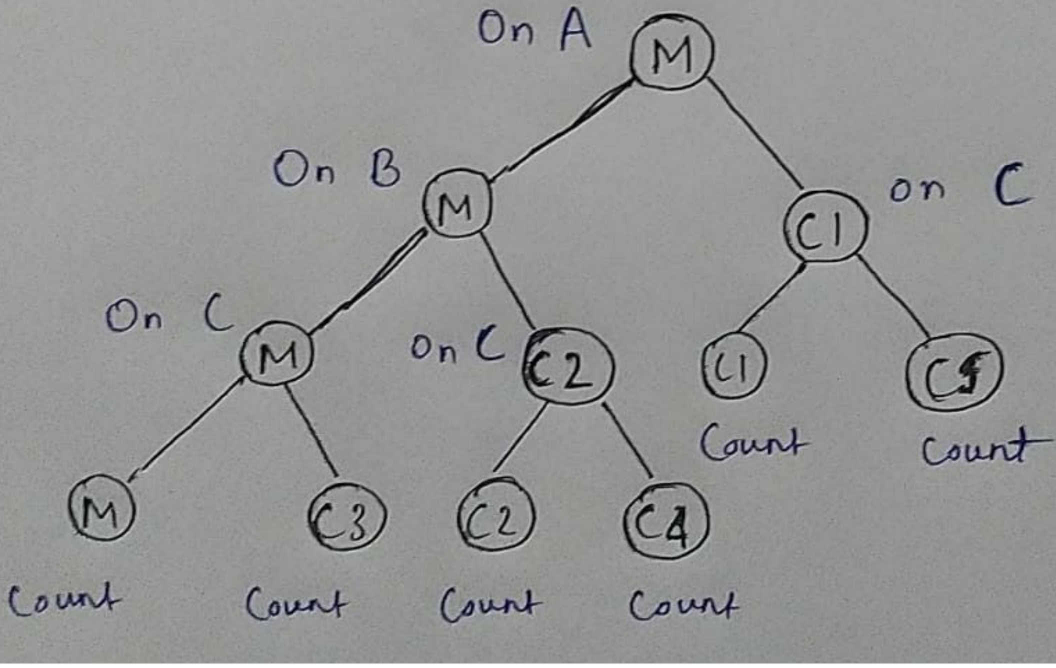
```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ gcc code1.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
Count
Count
Count
Count
Count
Count
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ 
```

Manual trace:

M used to denote ***main()***
C1, C2, C3, C4, C5 are used to denote ***children***

Count will be printed **6** times.

```
int main ( )
{
    pid_t pid;
    pid = fork ( ); // A
    if (pid != 0)
    fork ( ); // B
    fork ( ); // C
    printf ("Count\n");
    return 0;
}
```



Reasoning:

On A, M (main) will split into two processes, M and C1 (or, will give rise to a new process C1), where C1 is the child of M. Assuming that fork doesn't fail (i.e. pid cannot be less than 0), pid ≠ 0 can only mean that pid > 0 i.e. control is with the parent. So, only M can access the if block and C1 cannot.

M on B again splits into two processes, M and C2 (or, gives rise to a new process C2), where C2 is the child of M.

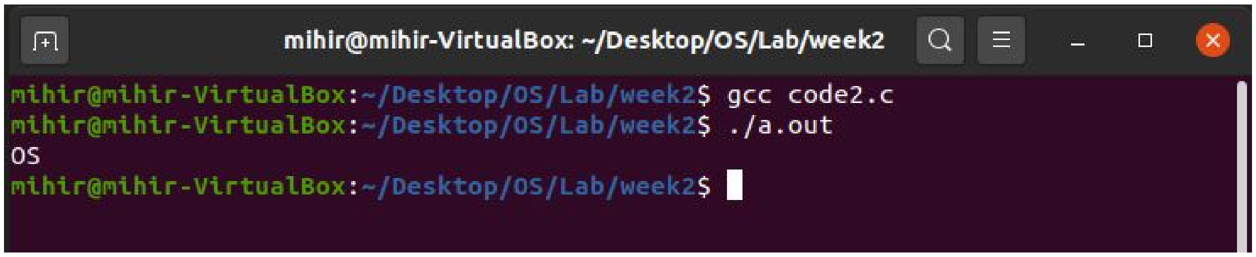
Now, each one of M, C1 and C2 can access C (the last fork statement). So, M splits into M and C3 (C3 being the child of M), C2 splits into C2 and C4 (C4 being the child of C2) while C1 splits into C1 and C5 (C5 being the child of C1).

Finally, each one of M, C1, C2, C3, C4, C5 and C6 can access the printf statement (since it is not inside any if block) and hence, **Count** is printed **6** times.

Question 2.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
        printf("OS \n");
        fork();         //A
        fork();         //B
        fork();         //C
        return 0;
}
```

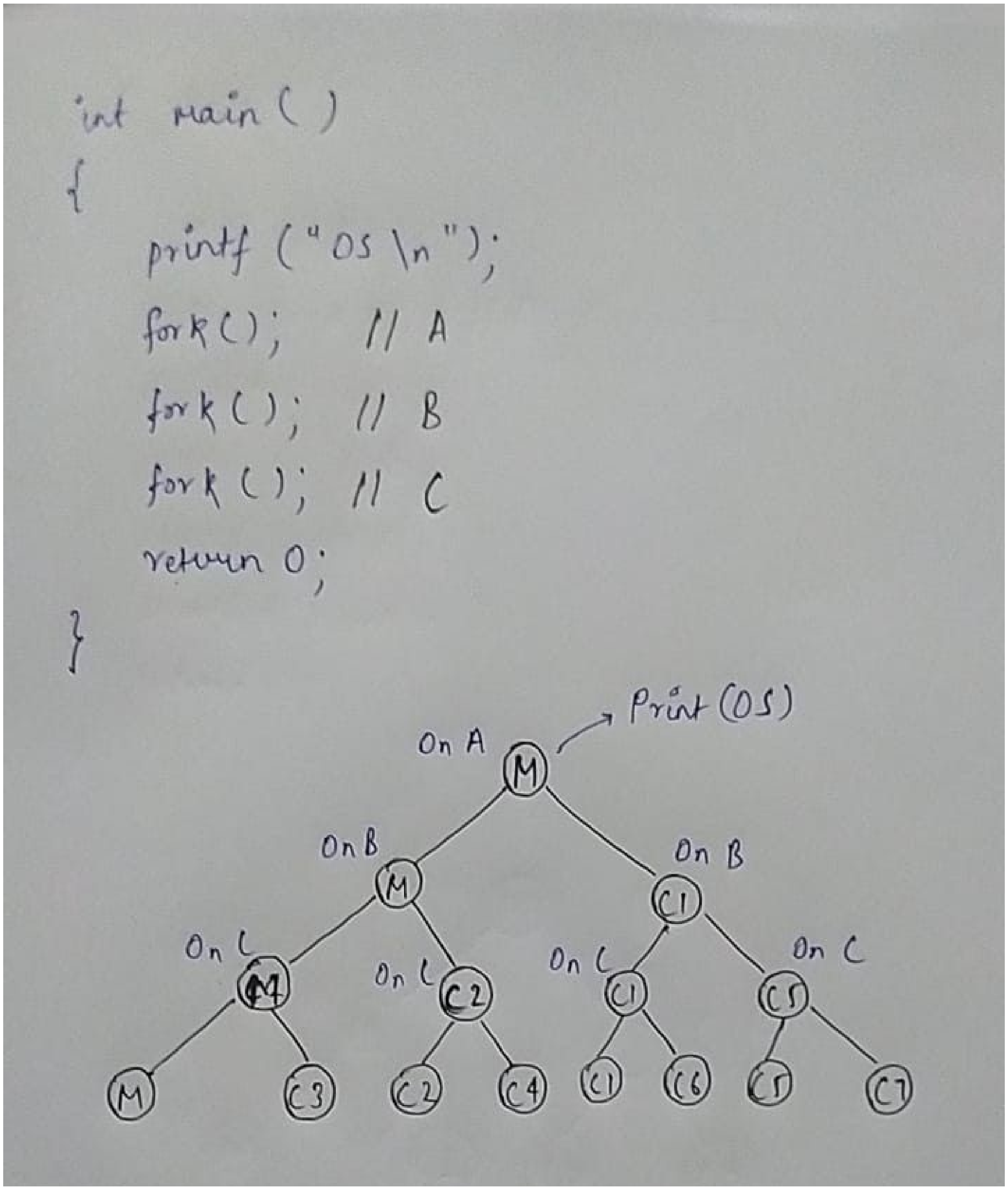
Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ gcc code2.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
OS
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$
```

Manual trace:

OS will be printed **1** time.



```
int main ( )
{
    printf ("OS \n");
    fork();     // A
    fork();     // B
    fork();     // C
    return 0;
}
```

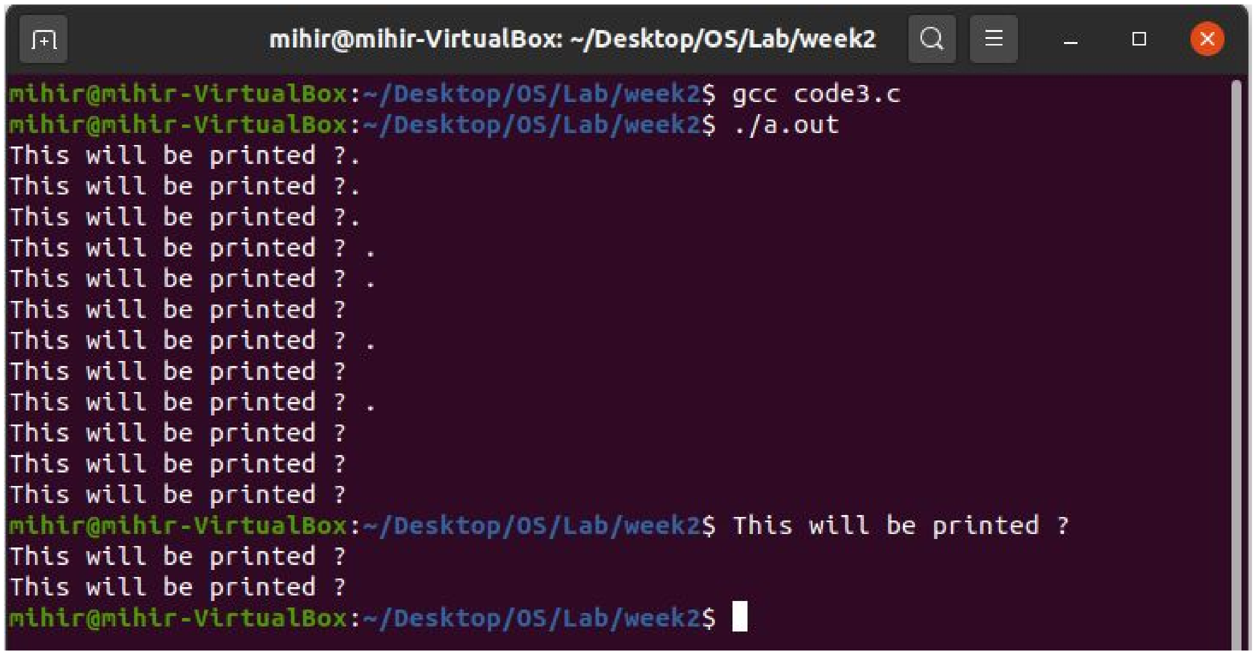
Reasoning:

Since the printf statement is above the very first fork() call, hence any of the child processes created after that will not be able to access the printf statement and it will only be accessed once by the first main() call (./a.out). So, **OS** will be printed only **once**.

Question 3

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main ()
{
        printf("This will be printed ?.\n");
        fork();         //A
        printf("This will be printed ?.\n");
        fork();         //B
        printf("This will be printed ? .\n");
        fork();         //C
        printf("This will be printed ?\n");
        return 0;
}
```

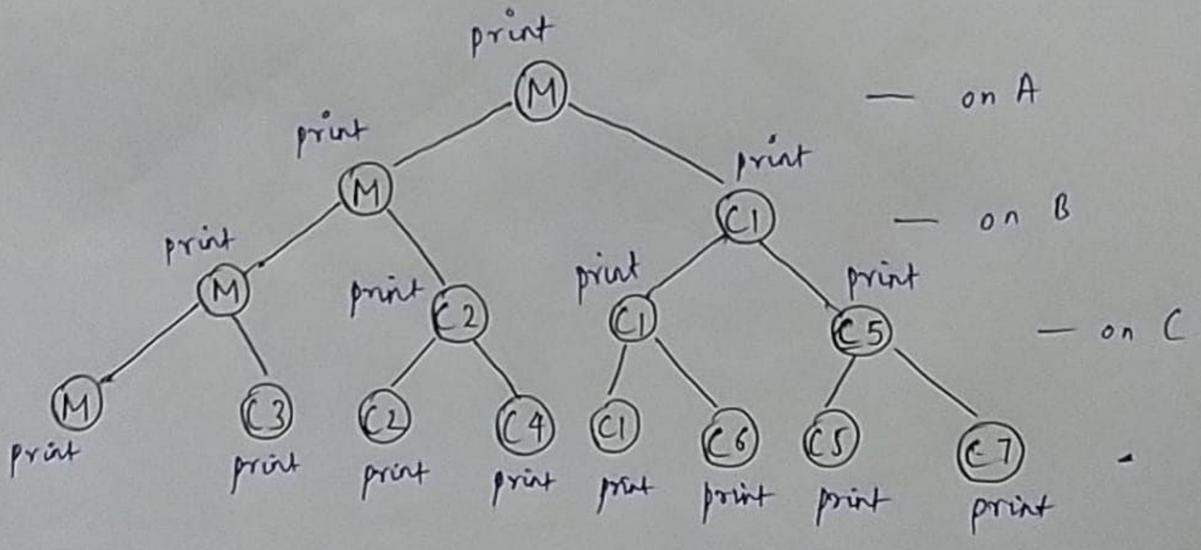
Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ gcc code3.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
This will be printed ?.
This will be printed ?.
This will be printed ?.
This will be printed ? .
This will be printed ? .
This will be printed ?
This will be printed ? .
This will be printed ?
This will be printed ? .
This will be printed ?
This will be printed ?
This will be printed ?
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ This will be printed ?
This will be printed ?
This will be printed ?
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$
```

Manual trace:
Total number of prints will be **15**.

```
int main ( )
{
    printf ("This will be printed?\n");
    fork ( );    // A
    printf ("This will be printed? \n");
    fork ( );    // B
    printf ("This will be printed? \n");
    fork ( );    // C
    printf ("This will be printed? \n");
    return 0;
}
```



Reasoning:

On calling M (main() or a.out), the first printf statement will be printed **(1)**.

On A, M splits into two processes, M and C1, (or gives rise to a new process C1) where C1 is the child of M. Then, M prints the second statement **(2)** and moves forward to B. On B, M again splits into two processes, M and C2, where C2 is the child of M. Then, M prints the third statement **(3)** and moves forward. On C, M again splits into two processes, M and C3, where C3 is the child of M. Then, M prints the last printf statement **(4)**.

For C1, the program starts from the statement after //A. It prints the second printf statement **(5)** and moves forward to B. On B, C1 splits into C1 and C5 (or gives rise to a new process C5), where C5 is the child of C1. Then, C1 prints the third printf statement **(6)** and moves forward to C. On C, C1 again splits into two processes, C1 and C6, where C6 is the child of C1. Then, C1 prints the last printf statement **(7)**.

For C2, the program starts from the statement after //B. It prints the third printf statement **(8)** and moves forward to C. On C, C2 splits into C2 and C4 (or gives rise to a new process C4), where C4 is the child of C2. Then, C2 prints the final printf statement **(9)**.

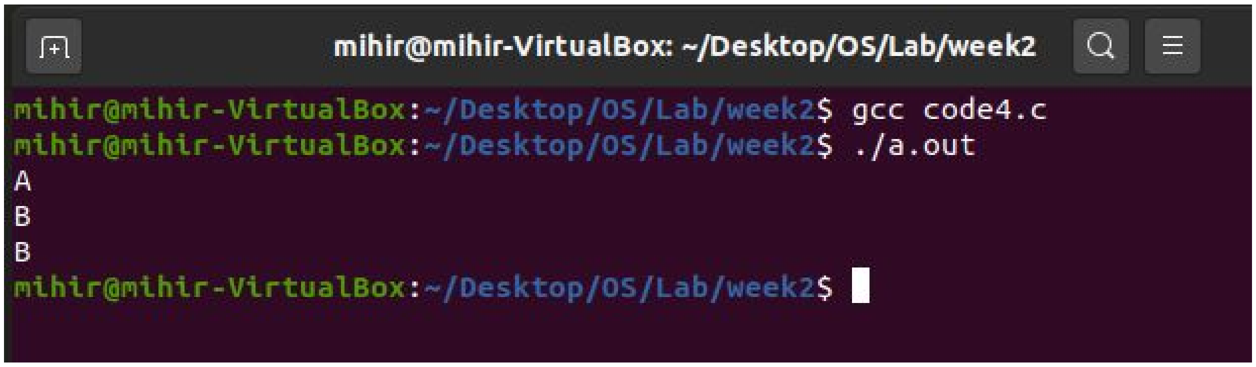
For C5, the program starts from the statement after //B. It prints the third printf statement **(10)** and moves forward to C. On C, C5 splits into C5 and C7 (or gives rise to a new process C7), where C7 is the child of C5. Then, C5 prints the final printf statement **(11)**.

For C3, C4, C6 and C7, the last printf is the only statement which they will execute since they were created at C. So, each one of them will print the last printf statement **(12) (13) (14) (15)**.

Question 4

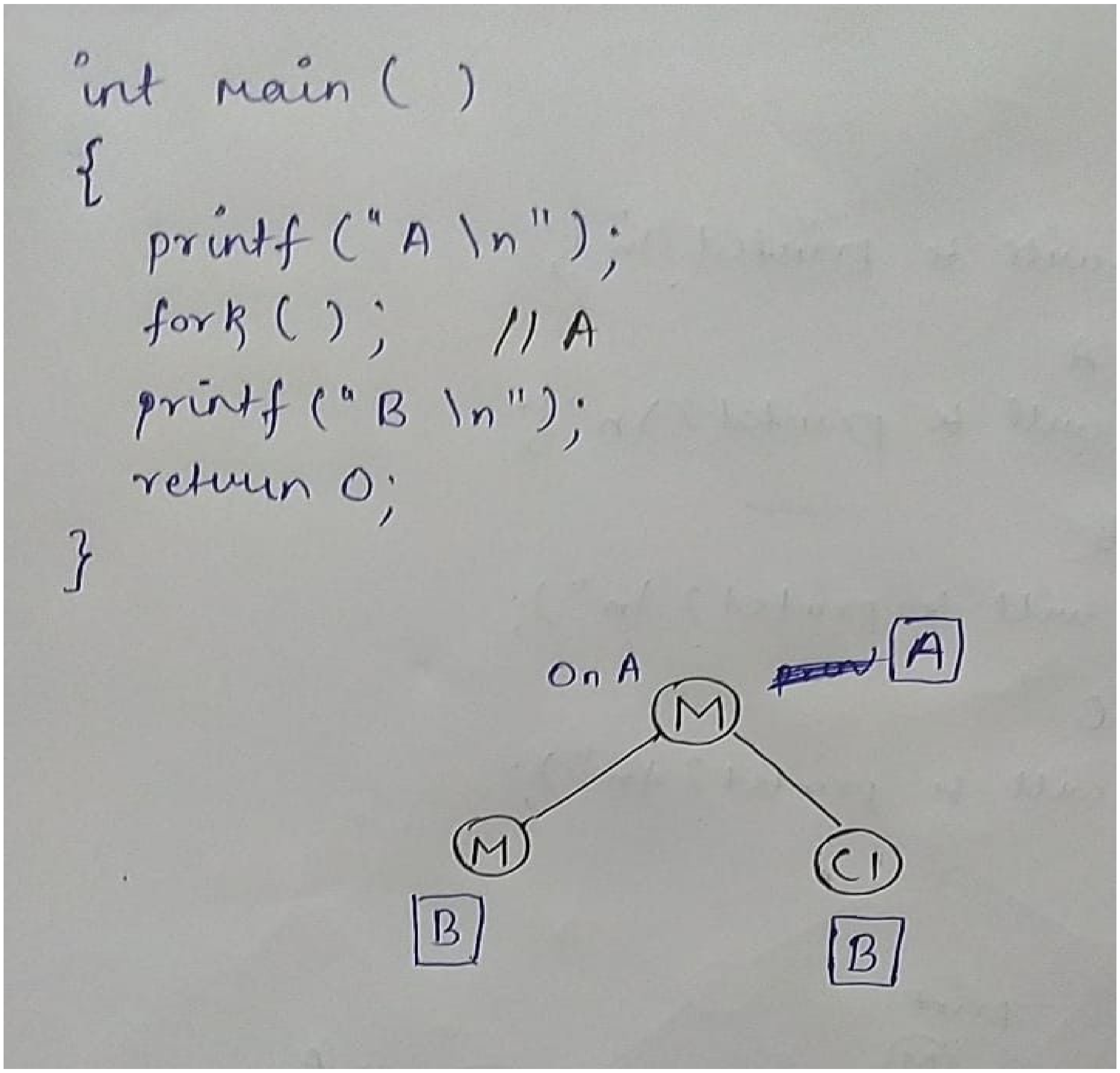
```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main ()
{
        printf("A \n");
        fork();
        printf("B\n");
        return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ gcc code4.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
A
B
B
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ 
```

Manual trace:



```
int main ( )
{
    printf ("A \n");
    fork ( );      // A
    printf ("B \n");
    return 0;
}
```

Output will be **ABB**.

Reasoning:

On calling M (main() or a.out), the first printf statement will be executed i.e. A **(1)** will be printed.

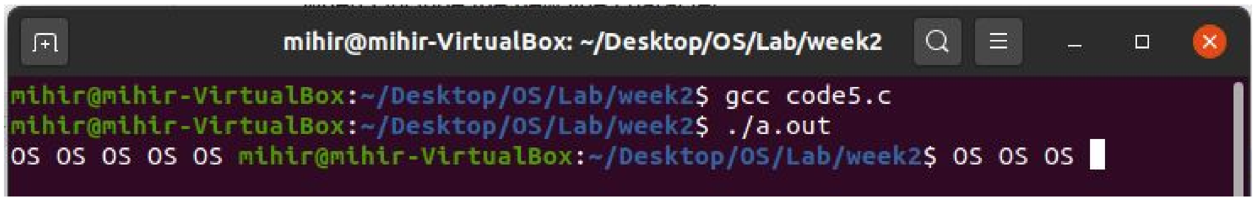
On A, M splits into M and C1. M continues and prints B **(2)**. For C1, the program starts from printf("B\n"); So, B **(3)** will be printed.

Hence, **A** will be printed **1** time and **B** will be printed **2** times and output will be **ABB**.

Question 5

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
        printf("OS ");
        fork();
        fork();
        fork();
        return 0;
}
```

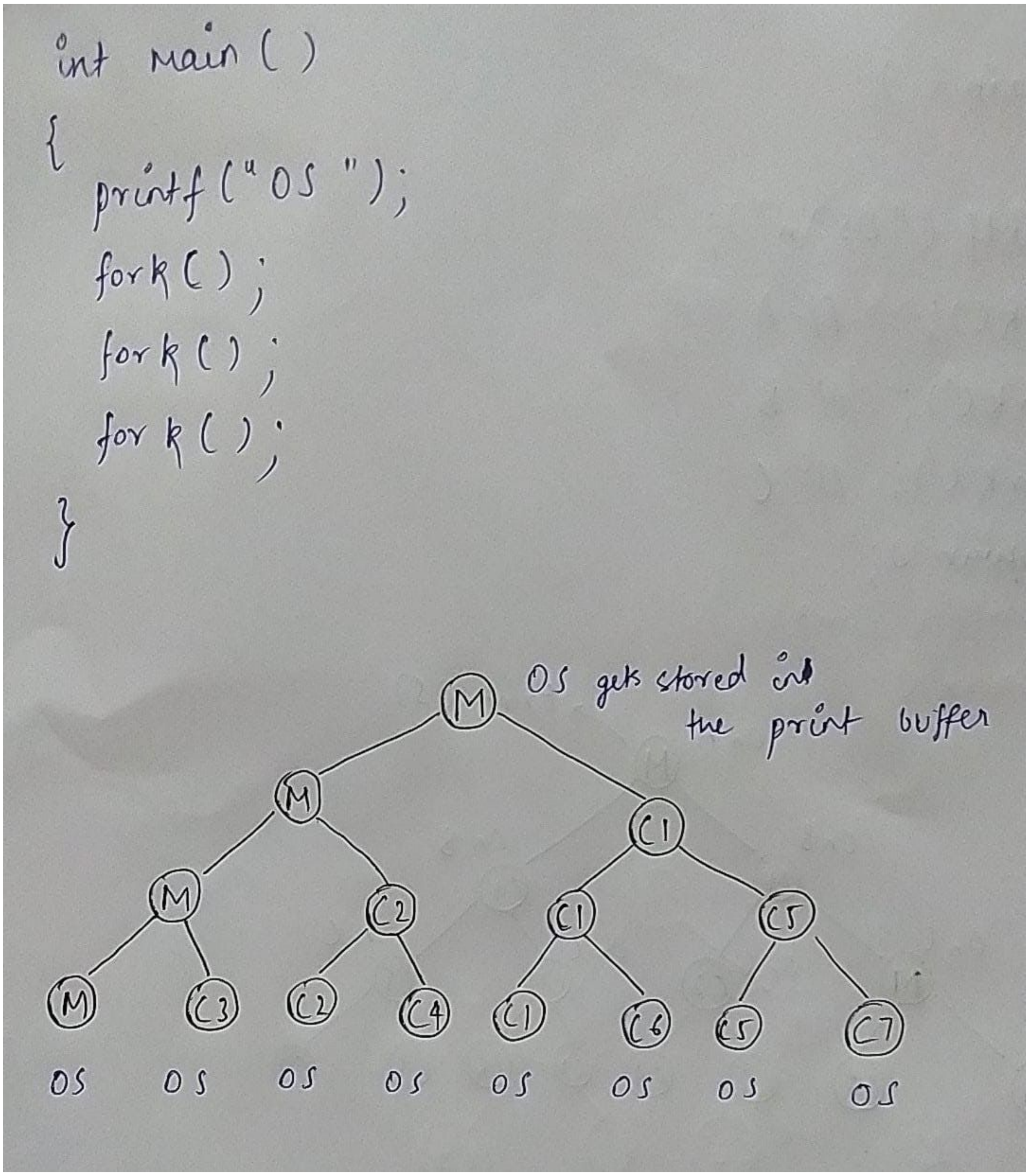
Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ gcc code5.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
OS OS OS OS OS mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ OS OS OS
```

Manual trace:

OS will be printed **8** times (space separated).



```
int main ( )
{
    printf ("OS ");
    fork ( );
    fork ( );
    fork ( );
}
```

Reasoning:

Notice that there is no \n (newline character) at the end of the printf statement.

When outputting to standard output using the C library's printf() function, the output is usually buffered. The buffer is not flushed until you output a newline, call fflush(stdout) or exit the program.

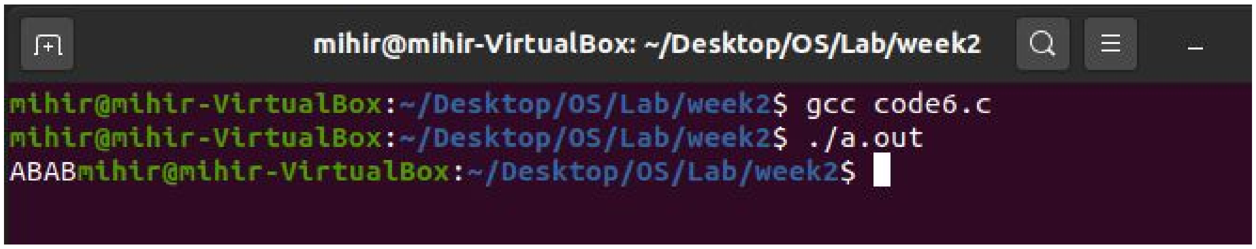
Initially, when we call main() and it executes the first printf statement, **OS** gets stuffed into the print buffer.

When we fork the process, the child process inherits every part of the parent process, including the unflushed output buffer. This effectively copies the unflushed buffer to each child process and since there are 7 child processes and a main() so, the print buffer already containing **OS** will be copied to all the child processes and **OS** will be printed **8** times separated by white space.

Question 6

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main ()
{
        printf("A");
        fork();
        printf("B");
        return 0;
}
```

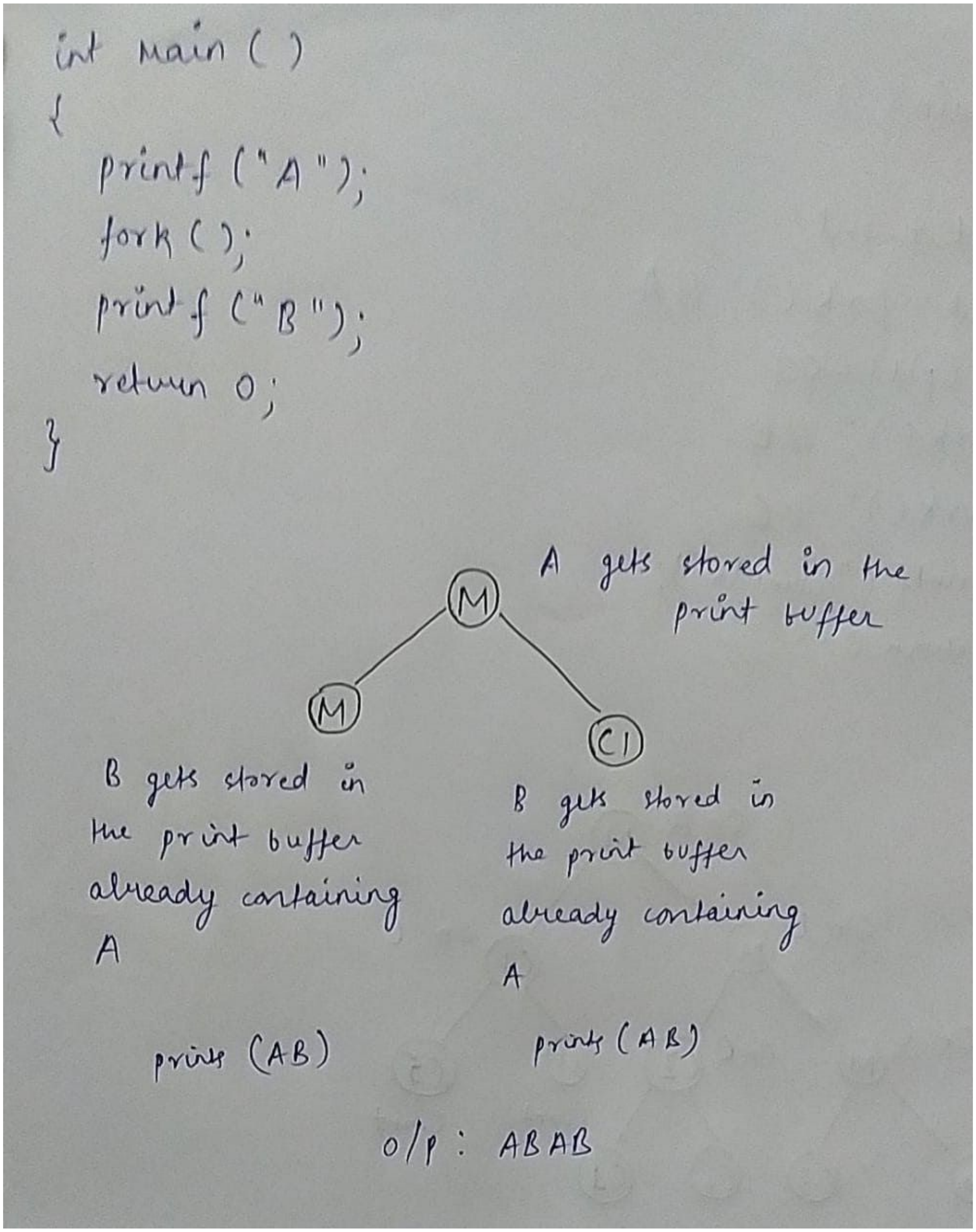
Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week2
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ gcc code6.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
ABABmihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$
```

Manual trace:

Output will be **ABAB**.



```
int main ( )
{
   printf ("A");
   fork ( );
   printf ("B");
   return 0;
}
```

A gets stored in the print buffer

B gets stored in the print buffer already containing A

prints (AB)

B gets stored in the print buffer already containing A

prints (AB)

o/p : ABAB

Reasoning:

Notice that there is no \n (newline character) at the end of the printf statements.

When outputting to standard output using the C library's printf() function, the output is usually buffered. The buffer is not flushed until you output a newline, call fflush(stdout) or exit the program.

Initially, when we call main() and it executes the first printf statement, **A** gets stuffed into the print buffer.

When we fork the process, the child process inherits every part of the parent process, including the unflushed output buffer. This effectively copies the unflushed buffer to each child process.

Now, when M completes its execution by executing the last printf statement, **B** gets stuffed to the print buffer already containing **A**. So, **AB** is printed

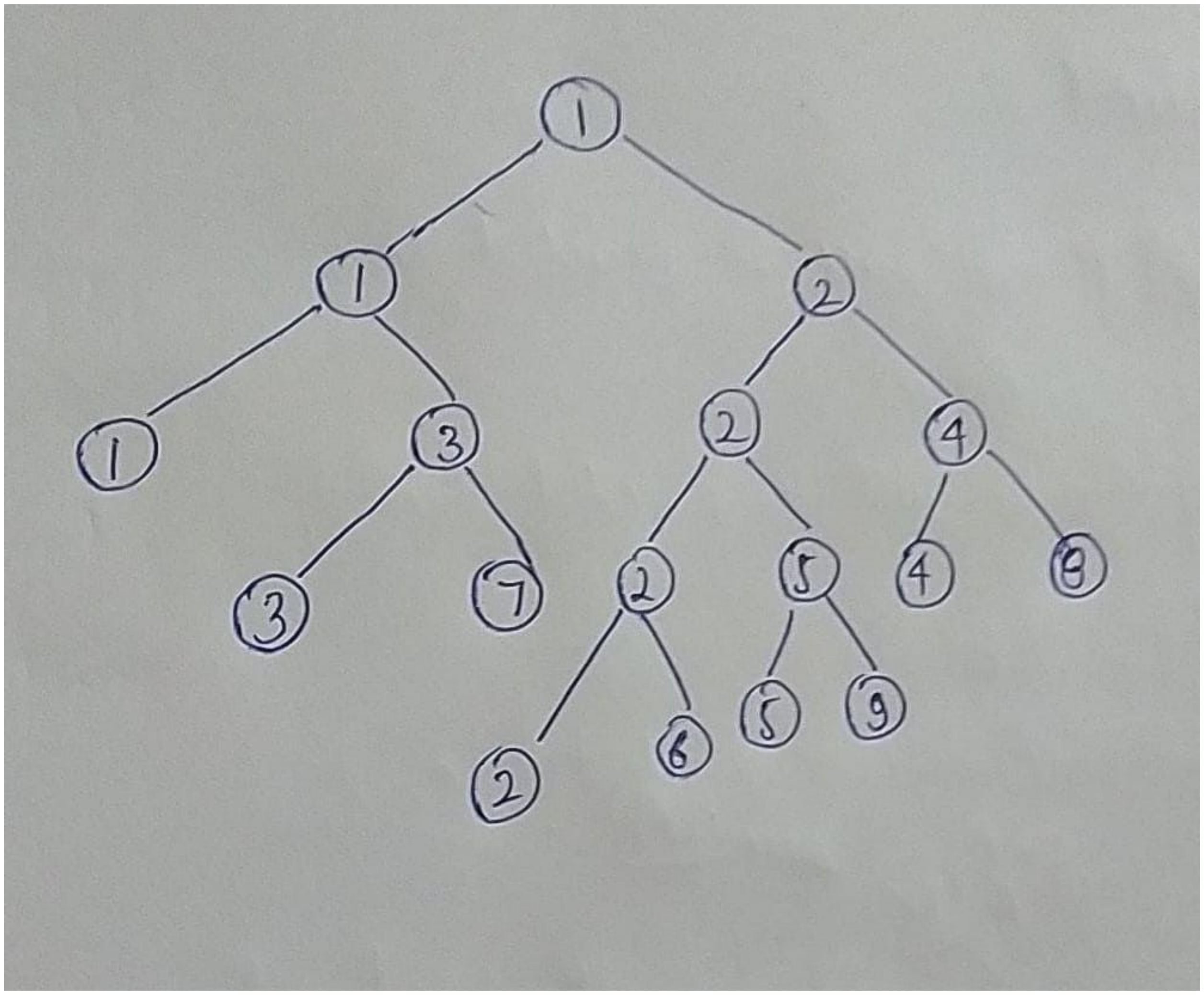
When C1 starts its execution i.e. when C1 executes the last printf statement, since it shares the same print buffer as its parent process (Note that the output buffer of the parent already has **A** stuffed into it) and **B** gets added to it. So, C1 also prints **AB**.

Since, no printf statement had \n or white space, final output will be **ABAB**.

Question 7

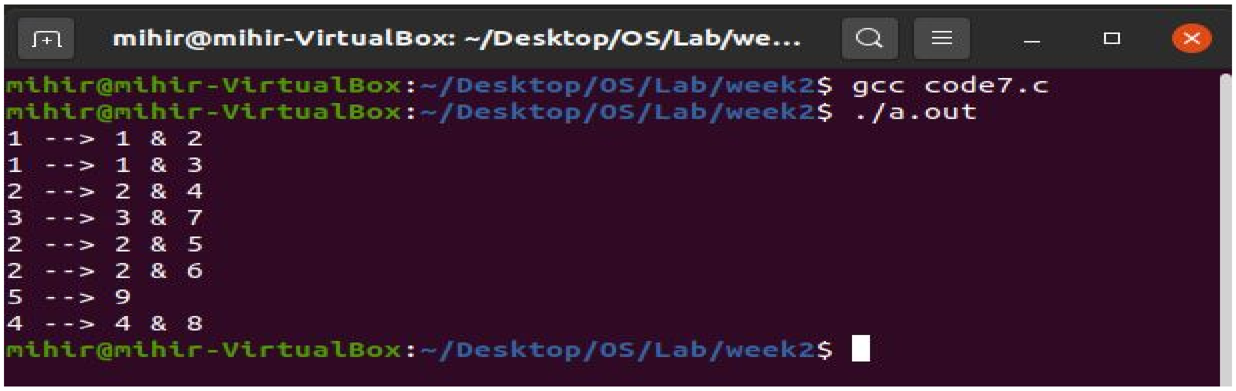
Express the following in a process tree setup and also write the C code for the same setup
1 forks 2 and 3
2 forks 4 5 and 6
3 forks 7
4 forks 8
5 forks 9

Process tree:



C code:

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main()
{
    pid_t pid;
    printf("1 --> 1 & 2\n");
    pid = fork();
    if (pid == 0)
    {
            pid_t pid1;
            printf("2 --> 2 & 4\n");
            pid1 = fork();

            if (pid1 == 0)
            {
                        printf("4 --> 4 & 8\n");
                        fork();
            }

            else if (pid1 > 0)
            {
                        printf("2 --> 2 & 5\n2 --> 2 & 6\n5 --> 9\n");
                        fork();
                        fork();
            }
    }

    else if (pid > 0)
    {
            pid_t pid2;
            printf("1 --> 1 & 3\n");
            pid2 = fork();

            if (pid2 == 0)
            {
                        printf("3 --> 3 & 7\n");
                        fork();
            }
    }

    return 0;
}
```
Output ss:



mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/we...

```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ gcc code7.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ ./a.out
1 --> 1 & 2
1 --> 1 & 3
2 --> 2 & 4
3 --> 3 & 7
2 --> 2 & 5
2 --> 2 & 6
5 --> 9
4 --> 4 & 8
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week2$ 
```