Adhesh Reghu Kumar
COE18B001

Operating Systems Lab

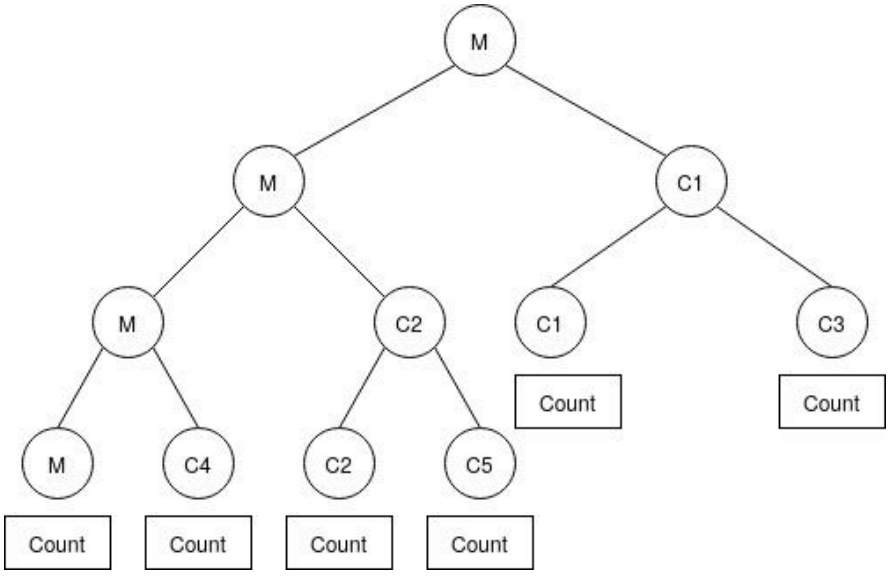
Lab - 2

Fork Practice Set

1.

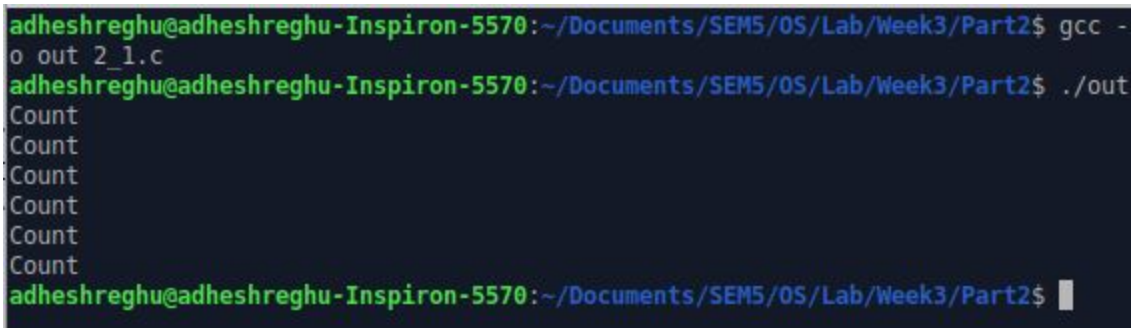
```c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

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

Manual Trace:



The above process tree represents the given program. So there are a total of 6 processes that will get executed (1 Main + 5 child processes). The *printf()* statement is a part of all the processes, hence will be executed 6 times. Hence “Count” will be printed 6 times (each time on a newline).

Output:



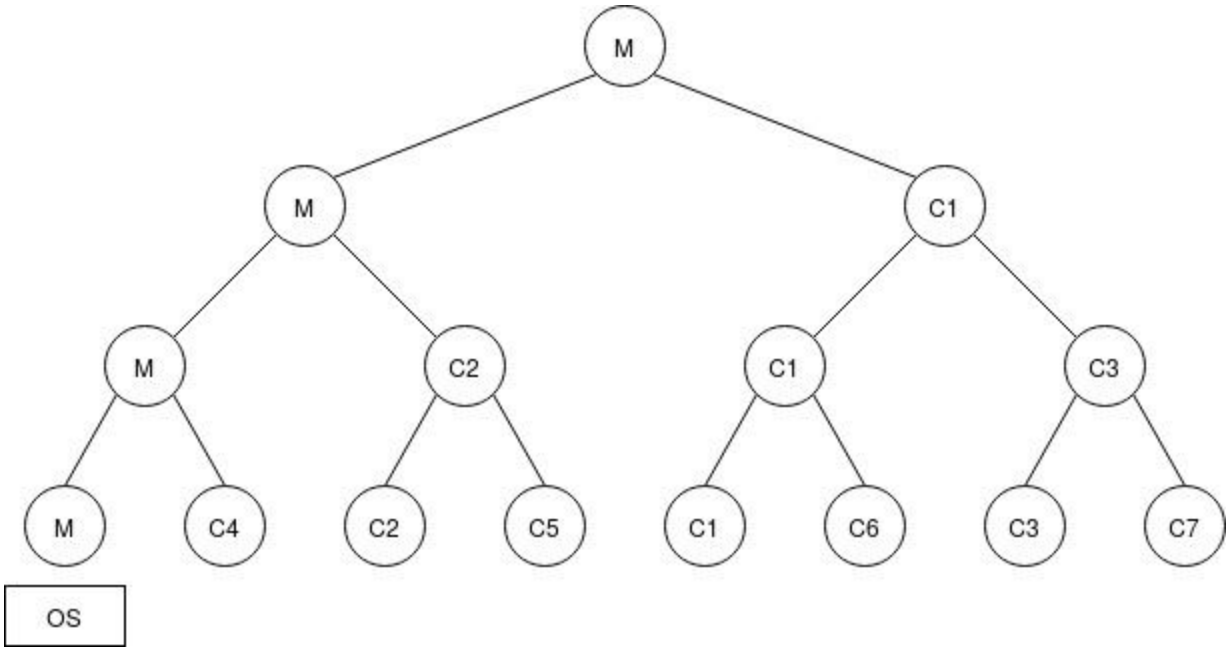
Reasoning/Understanding

The *main* is initially forked once and the *pid* is stored in a variable called pid. So M gives rise to child process *C1*. The pid = 0 when the child process executes and pid > 0 when parent executes.
The if block condition satisfies that of the parent, hence the parent main (M) gets into the if block and is forked once more giving one more child process *C2*.
The third fork does not fall under the if block and is a part of the normal main block, so would be executed in both the parent and child processes. So both *M* and *C1* are forked once more. *C1* on being forked produces child *C3* and M on being forked produces *C4*. *C2* which is the child of the main forked the second time also gets forked and produces child process *C5*.

2.

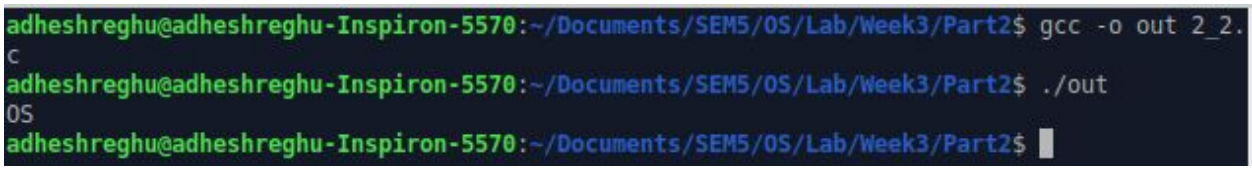
```c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

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

Manual Trace:



The above tree represents the trace of the given program. Since the forks() are called serially within the main, 7 children are created. Therefore a total of 8 processes (1 Main + 7 children) get executed. Since the *printf("OS \n")* stmt is executed in the beginning before the fork calls, therefore it gets executed only as a part of the main process. Hence it gets printed only once.

Output:



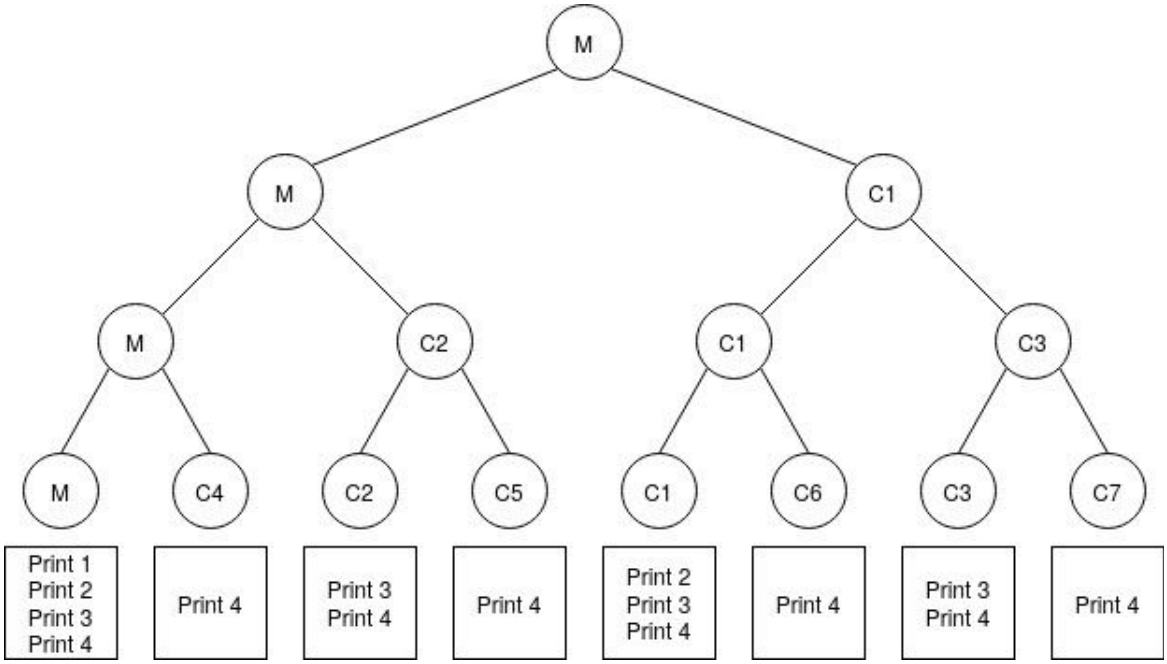
Reasoning/Understanding:

Serial calls of fork() within the main produces a total of 2^n processes. The main gets forked thrice producing children *C1*, *C2* and *C4* in the three fork calls respectively. *C1* gets forked twice and produces children *C3* and *C6*. Children *C2* and *C3* are only forked once and produce children *C5* and *C7* respectively.
As explained earlier, the printf gets executed only as a part of the *main* process and hence “OS” is only printed once.

3.

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

int main ()
{
printf("This will be printed once.\n");   // Print 1
fork();
printf("This will be printed twice.\n");   // Print 2
fork();
printf("This will be printed four times.\n");  // Print 3
fork();
printf("This will be printed eight times.\n");    // Print 4
return 0;
}
```

Manual Trace:



The above tree represents the given program. Since three forks are called serially within main, a total of 8 processes get executed. A child process executes all statements below the forking point. Hence accordingly the printf statements get executed.
Total prints executed = 4 + 1 + 2 + 1 + 3 + 1 + 2 + 1 = 15.

Output:

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$ ./out
This will be printed once.
This will be printed twice.
This will be printed four times.
This will be printed twice.
This will be printed four times.
This will be printed eight times.
This will be printed four times.
This will be printed eight times.
This will be printed eight times.
This will be printed four times.
This will be printed eight times.
This will be printed eight times.
This will be printed eight times.
This will be printed eight times.
This will be printed eight times.
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$
```

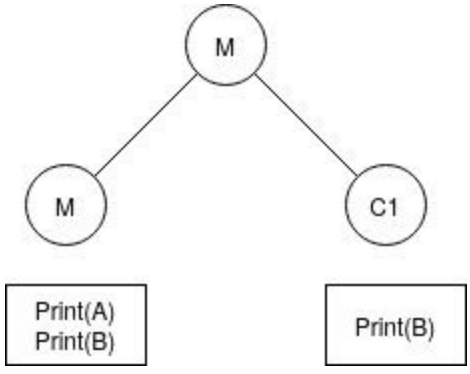
Reasoning/Understanding:

Main first prints *Print 1* statement. Then main (M) forks and produces a child *C1*. Both *M* and *C1* print *Print 2* statement. Then main (M) gets forked again to produce *C2*. While *C1* also gets forked to produce *C3*. Now *M, C1, C2* and *C3* all print *Print 3*. Now on the last fork M,C2,C1 and C3 get forked to produce children *C4, C5, C6* and *C7* respectively. Now *M, C1, C2, C3, C4, C5, C6* and *C7* print *Print 4.*

4.

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

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

Manual Trace:



The above tree represents the process tree of the given program execution. The main (*M*) forks and produces child *C1*.

Output:

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$ gcc -o out 2_4.
c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$ ./out
A
B
B
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$
```

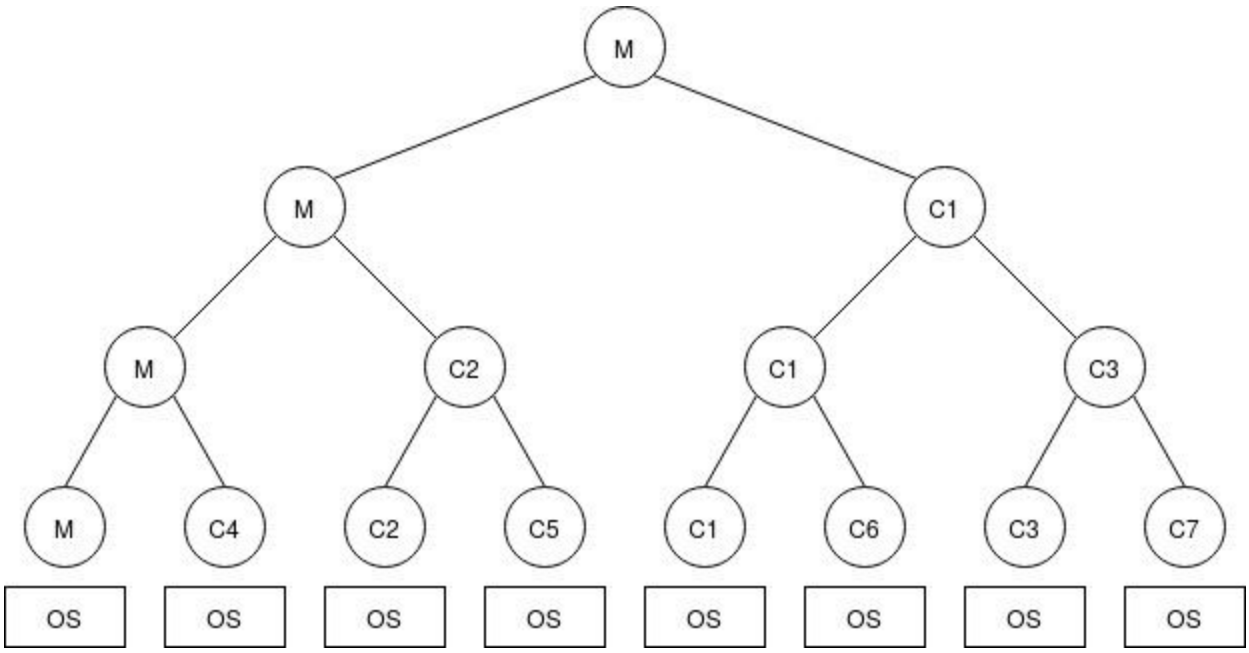
Reasoning/Understanding:

The first printf gets executed as part of the main (*M*) process. Then at the forking point *M* produces a child *C1*. Since the second printf statement occurs after the forking point, hence it gets executed as a part of both the main *M* and the child *C1* processes.

5.

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

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

Manual Trace:



The above process tree represents the given program execution.

Output:

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$ gcc -o out 2_5.
c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$ ./out
OS OS OS OS OS OS OS OS adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week
3/Part2$
```

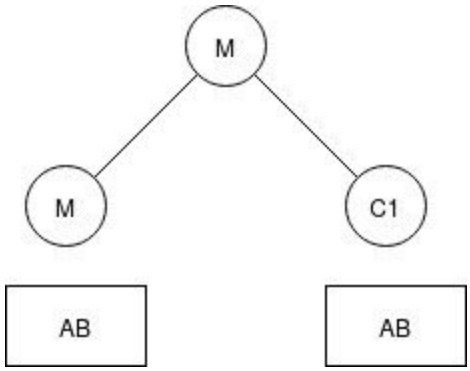
Reasoning/Understanding:

The process tree is explained in the above examples. The only difference in execution is the printing of *"OS"* 8 times. Why does this happen?
The *printf()* usually buffers its output. The buffer is not flushed until a newline or end of the program or call fflush.
So when we fork, the child processes inherit every part of the parent process, including the unflushed output buffer. This effectively copies the unflushed buffer to each child process. When the process terminates, the buffers are flushed. And hence we get 8 *prints* corresponding to the eight processes.

6.

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

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

Manual Trace:



The processes tree represents the given program execution. *M* forks to give child *C1*.

Output:

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$ gcc -o out 2_6.
c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$ ./out
ABABadheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week3/Part2$
```

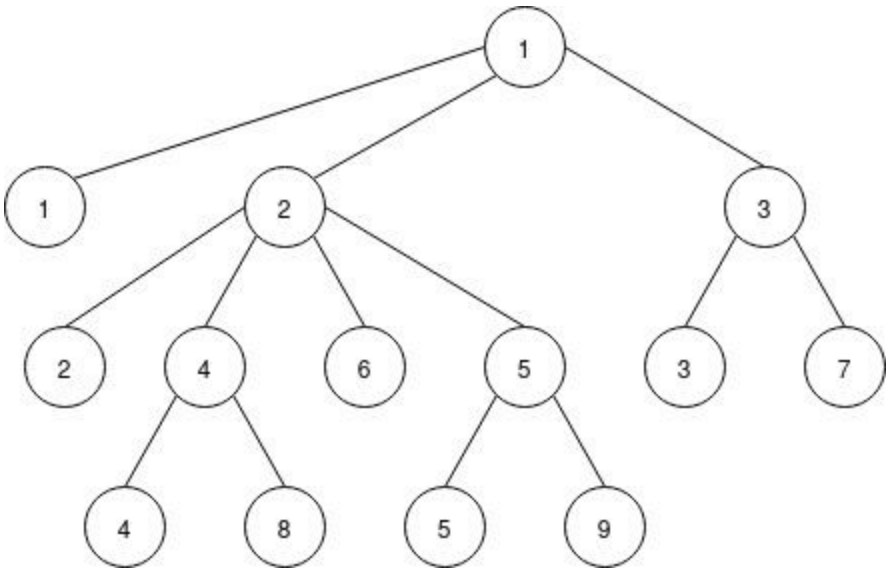
Reasoning/Understanding:

Similar to the last execution, since the printf does not have a newline in it, the buffer gets copied to the child process as well. After that child prints “B” to the buffer. At the end of the process termination, the child process prints the entire buffer “AB” (“A” inherited from the parent main’s output buffer) to stdout. The main anyway does print “AB”. Hence “ABAB” gets printed.

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:



Code:

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

int main ()
{
    int pid[8];

    printf("Parent Process - 1. Process id = %jd\n",(intmax_t)getpid());

    pid[0] = fork();                    // create process 2
    if(pid[0] == 0)
    {
       printf("Process - 2. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
       pid[2] = fork();                 // create process 4
       if(pid[2] == 0)
       {
             printf("Process - 4. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
             pid[6] = fork();           // create process 8
             if(pid[6] == 0)
                   printf("Process - 8. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
       }
       else if(pid[2] > 0)
       {
             pid[3] = fork();           // create process 5
             if(pid[3] == 0)
             {
                   printf("Process - 5. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
                   pid[7] = fork();     // create process 9.
                   if(pid[7] == 0)
                         printf("Process - 9. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
             }
             else if(pid[3] > 0)
             {
```

```
                pid[4] = fork();            // create process 6
                if(pid[4] == 0)
                      printf("Process - 6. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
            }
        }
    }
    else if(pid[0] > 0)
    {
        pid[1] = fork();                    // create process 3
        if(pid[1] == 0)
        {
              printf("Process - 3. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
              pid[5] = fork();              // create process 7
              if(pid[5] == 0)
                    printf("Process - 7. Process id = %jd, Parent Process =
%jd\n",(intmax_t)getpid(),(intmax_t)getppid());
        }
    }
}
```

Output:

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week2/Part2$ gcc -o out 2_7.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week2/Part2$ ./out
Parent Process - 1. Process id = 3568
Process - 2. Process id = 3569, Parent Process = 3568
Process - 3. Process id = 3570, Parent Process = 3568
Process - 4. Process id = 3571, Parent Process = 3569
Process - 7. Process id = 3573, Parent Process = 3570
Process - 5. Process id = 3572, Parent Process = 3569
Process - 6. Process id = 3574, Parent Process = 3569
Process - 8. Process id = 3575, Parent Process = 3571
Process - 9. Process id = 3576, Parent Process = 3572
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week2/Part2$
```