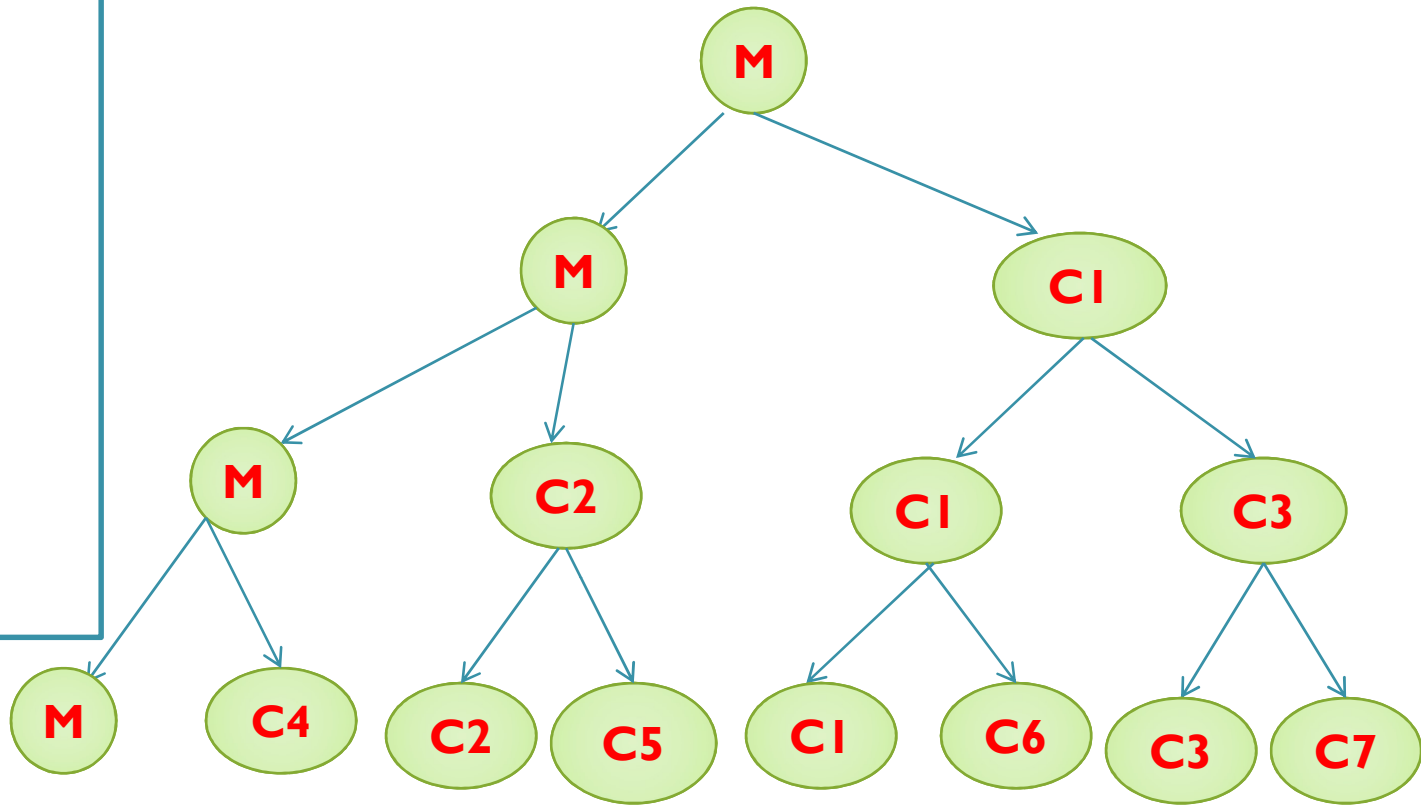
The classical FORK Bomb!

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



✓Overall 8 processes in Main Memory as indicated at the leaf level nodes count

✓In general n fork class (non conditional) will result in 2^n processes

```
fork ();    // Line 1
fork ();   // Line 2
fork ();   // Line 3

     L1       // There will be 1 child process created by line 1
   /     \
  L2      L2    // There will be 2 child processes created by line 2
 /  \    /  \
L3  L3  L3  L3  // There will be 4 child processes created by line 3
```

- By recurrence setup;
- $T(n) = 2T(n-1)$; $T(1) = 2$;
- $T(2) = 2T(1) = 4$;
- In general n forks results **in 2^n processes**

EXEC calls to overlay process images

✓So Far fork example we did, child process carried the same image "
as the parent process. Practicality requires child to have new definition.

✓**Is it possible ? Yes** and that's the purpose of the exec system calls!

✓Motive of forking – **simulate / achieve multiple / parallel processing** or execution of program(s)

✓Either different programs or same programs different portions!

✓Best Example – Word – As you type (edit) ; spell check happens! . Can view as multiple processes – but in OS parallelism is best achieved with Threading!

First Example of exec - execl

```
# include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
pid_t pid;   // this is to use the pid data type – relevant headers above
pid = fork();
if (pid == 0)
execl("/bin/ls","ls", NULL);  // child image is now ls command
else
{
wait (NULL);   // parent waits for the child to complete execution.
printf("Parent Process gets the control \n");
printf ("Parent Has waited for Child to Complete");
}
} Note: execl("/bin/ls","ls","-l", NULL); options comma separated list
```

- **wait()** system call suspends execution of the calling process until one of its children terminates
- **wait()**: on success, returns the process ID of the terminated child; on error, -1 is returned
- **waitpid()** suspends execution of the calling process until a child specified by *pid* argument has changed state. By default, **waitpid()** waits only for terminated children, but this behavior is modifiable via the *options* ; *wait(&status)* is equivalent to:waitpid(-1, &status, 0);
- < -1 meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
- -1 meaning wait for any child process.
- > 0 meaning wait for the child whose process ID is equal to the value of *pid*.