Calling `fork()` once creates a child process which essentially creates a **separate memory space**. At the time of fork(), both processes have the same content.

Every call leads to an increase in the memory that is occupied. In a practical situation, our computer has limited main memory. Hence, creating a process everytime leads to a lot of **overhead**, thus causing the `fork()` call to fail[1].

In most operating systems, there is a limit on the number of processes that can run on a pc. This can be checked using the command : `ulimit -u`

In our case, an output of `24000` was obtained. This depends on a variety of factors with the available **RAM** being one of the primary factors.

On testing the code with larger number of rows and columns, the fork() call failed when it was called from the parent for the 16250th time (on an average) with `errno = EAGAIN` [2] . Since we are creating as many as `r1 * c2` processes, we obtain a bound on the product.

Interestingly, the number of allowed processes are significantly higher than the total number of fork() calls that are actually made in the program. This is because there are other processes that are being handled by the kernel at the same time.

Apart from this the concurrency was also measured using a counter variable to see how many processes actually execute concurrently.

For fixed c1 (r2)

r1 * c2	c1 (r2)	Max concurrency
1000	100	4
3000	100	6
10000	100	7

For a given value of r2 (or c1), smaller values of r1 * c2, leads to lower number of concurrent processes. This is because there arent many children to fork in the first place.

For fixed r1 * c2

r1 * c2	c1 (r2)	Max concurrency
300	10	4
300	50	6
300	300	6

For a given value of r1 * c2, smaller values of r2 (or c1), leads to lower number of concurrent processes. This is because there are less context switches owing to the fact that the CPU burst time for each process decreases with decrease in r2 (or c1).

These statistics also depend on the **number of cores** that are available to us. The maximum speedup would be equal to the number of cores, however in practical cases this would be less than the number of cores since there are many other processes running in the system.

References

As per man page for fork(),

EAGAIN : It was not possible to create a new process because the caller's RLIMIT_NPROC resource limit was encountered. To exceed this limit, the process must have either the CAP_SYS_ADMIN or the CAP_SYS_RESOURCE capability.

EAGAIN : A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

- the RLIMIT_NPROC soft resource limit (set via setrlimit(2)), which limits the number of processes and threads for a real user ID, was reached;
- the kernel's system-wide limit on the number of processes and threads, /proc/sys/kernel/threads-max, was reached (see proc(5));
- the maximum number of PIDs, /proc/sys/kernel/pid_max, was reached (see proc(5)); or
- the PID limit (pids.max) imposed by the cgroup "process number" (PIDs) controller was reached.

Fork can also fail when the almost all of the memory is occupied. The following flag will be raised in that case.

ENOMEM fork() failed to allocate the necessary kernel structures because memory is tight.

RLIMIT_NPROC This is a limit on the number of extant process (or, more precisely on Linux, threads) for the real user ID of the calling process. So long as the current number of processes belonging to this process's real user ID is greater than or equal to this limit, fork(2) fails with the error EAGAIN.

This functions sets up the max_threads limit for fork.

```
 void __init fork_init(unsigned long mempages)
{
    /*
     * The default maximum number of threads is set to a safe
     * value: the thread structures can take up at most half
     * of memory.
     */
    max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 8;

    init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
    init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
}
```