SHREY MARWAHA 2019334

NOTE: The output of both the .c files will contains result as **p**10, **p**33, **c**10, **c** -25 → in these c corresponds to '**child**' and p corresponds to '**parent**'. All the numbers contain such c and p to clarify the result obtained. Also the initial and final value of parent and child in both the cases is printed.

EXPLAINATION:

In part 1 (FORKING) we observe that the global variable (global_num) which was initialized to 10, the parent process increases global variable up to 100 serially (11,12,13,---100) from 10. And the child process decreases the global variable from 10 to -90 serially. Since fork() actually duplicates the process, so after calling fork there are actually 2 instances of your program running. The parent and child process are **completely independent** and **doesn't share any memory space (heap/stack),** each one has their own separate heap, stack and other memory space. So that's why the child process starts decreasing the global variable from 10 and not from 100 (as we see in the threads). The child process and parent cannot share any data until and unless some IPC is maintained between them.

In part2 (THREADS) we observed that the global variable named global_num which was initialized to 10, first it increases serially up to 100 (11,12,13,---100) and then decreases serially from the 100 only (and not from 10 as observed in forking) to -90 (100, 99, 98,--- 0, -1,---, -90), not like what we see in case of forking in which it started decreasing from 10, which is the initialized value of global variable.

This is because when a process starts, it is assigned memory and resources. Each thread in the process shares that memory and resources. There are two types of memory available to a process or a thread, the stack and the heap. It is important to distinguish between these two types of process memory because **each thread will have its own stack**, but all the threads in a process will **share the same heap**. Threads are therefor called lightweight processes because each thread have their own stack but can access shared data. Because threads share the same address space as the process and other threads within the process, the operational cost of communication between the threads is low, which is an advantage. The disadvantage is that a problem with one thread in a process will certainly affect other threads and the viability of the process itself. And because of that shared memory map ,and hence it share all the global data (static variables, global variables, and memory that is dynamically-allocated via malloc or new), is why we observe that the thread that we created using pthread_create will decrease the global variable from 100 and not from 10 (unlike forking).

Threads share the following:

(1) Data Segment(**global variables**, static data)

(2) Address space.

(3) Code Segment.

(4) I/O, if file is open, all threads can read/write to it.

(5) Process id of parent.

(6) The Heap

*NOTE: if we do same experiment with local variables then we would observe same output in thread and fork. This is because threads maintain their own copy of stack, and local variables are stored on the stack so each thread should have its own copy of local variables.