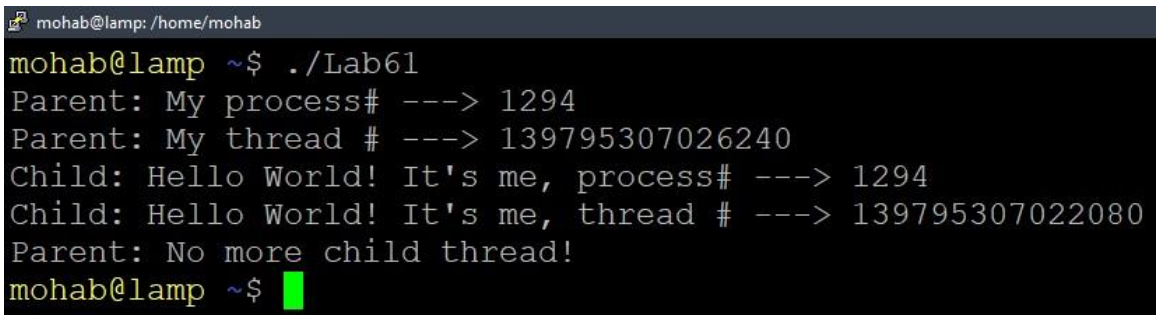
Example1 (thread creation and termination):

1) Run the above program, and observe its output:



2) Are the process ID numbers of parent and child threads the same or different? Why?

The process ID numbers of the parent and child threads will be the same. Because both the parent process and the child thread belong to the same process

Example2 (thread global data):

3) Run the above program several times; observe its output every time. A sample output follows:

4) Does the program give the same output every time? Why?

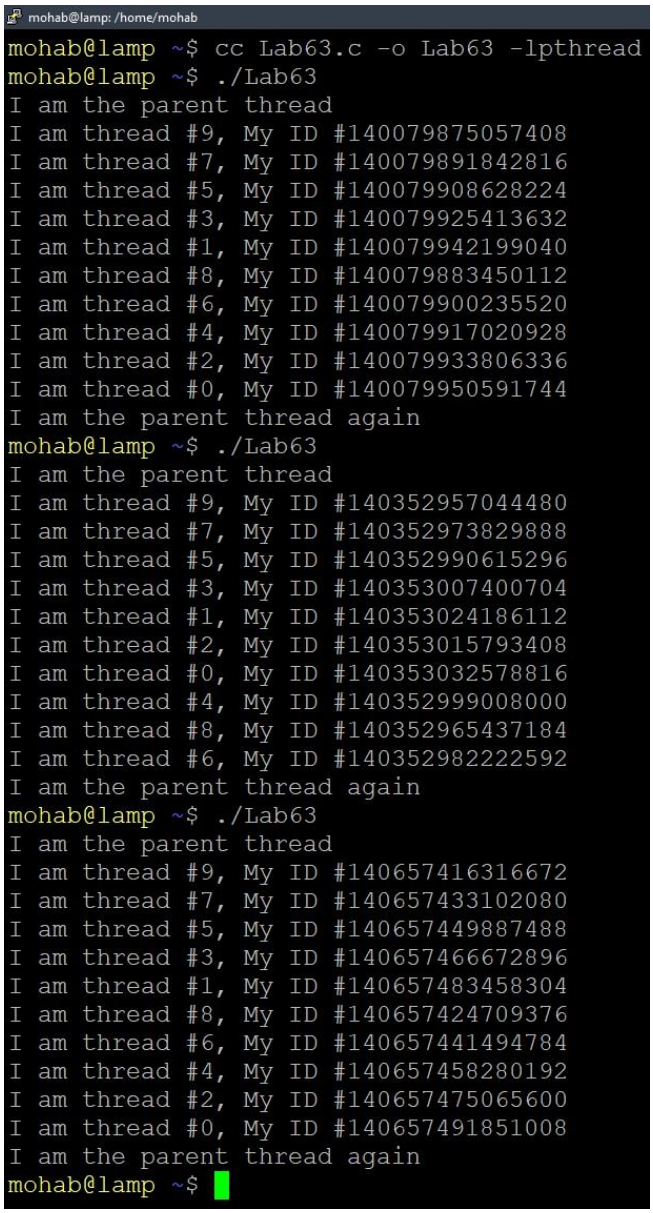
No, Because the program involves concurrent execution of multiple threads, and the order of execution is not deterministic.

5) Do the threads have separate copies of `glob_data`?

No, this is why the child thread's update to glob_data affects the value seen by the parent thread.

Example3 (multi-threaded program):

6) Run the above program several times and observe the outputs:

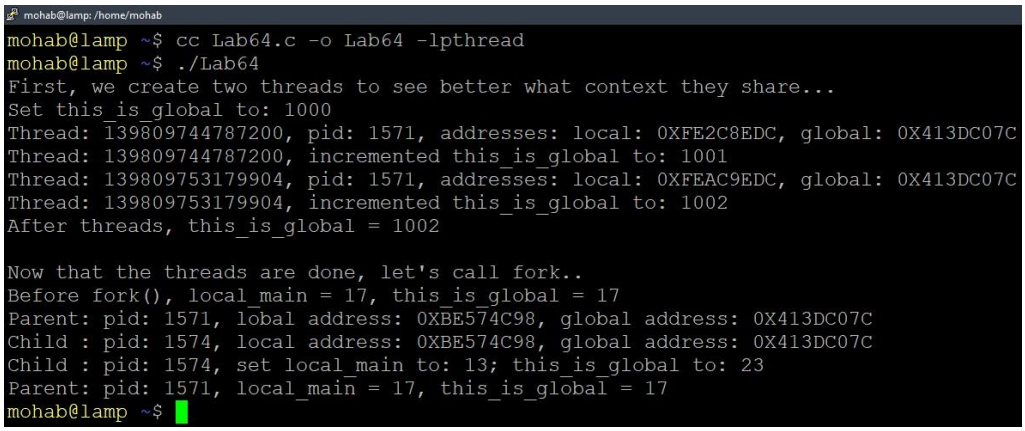


7) Do the output lines come in the same order every time? Why?

No, the order of the thread messages is not necessarily in numerical order because the operating system's scheduler determines which thread runs at any given time.

Example4 (difference between processes and threads):

8) Run the above program and observe its output. Following is a sample output:



9) Did `this_is_global` change after the threads have finished? Why?

Yes. This is because threads share the same memory space within a process, including global variables. Therefore, any modifications made by one thread to a shared global variable will be visible to other threads.

10) Are the local addresses the same in each thread? What about the global addresses?

The local addresses will be different in each thread. Each thread has its own stack, so their local variables are stored in different memory locations. However, the global addresses will be the same in each thread since they share the same memory space.

11) Did `local_main` and `this_is_global` change after the child process has finished? Why?

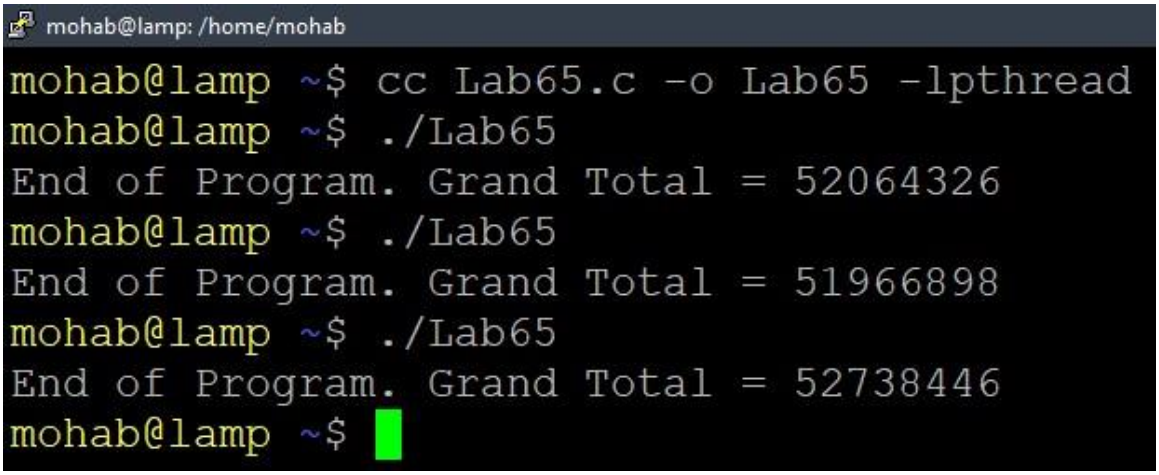
No. This is because processes have separate memory spaces, and changes made in one process do not affect the memory of other processes. The child process creates a copy of the parent's memory, including the values of the variables at the time of forking. Any changes made in the child process will not be reflected in the parent process.

12) Are the local addresses the same in each process? What about global addresses? What happened?

The local addresses will be different in each process. Each process has its own memory space, including its own stack where local variables are stored. Similarly, the global addresses will also be different in each process since they have separate memory spaces.

Example5:

13) Run the above program several times and observe the outputs, until you get different results.



14) How many times the line `tot_items = tot_items + *iptr;` is executed?

The line tot_items = tot_items + *iptr; is executed multiple times by each thread in a loop. The loop iterates 50,000 times for each thread. Since there are 50 threads created, the line is executed a total of 50 threads * 50,000 iterations = 2,500,000 times.

15) What values does `*iptr` have during these executions?

During each execution of the line tot_items = tot_items + *iptr;, the value of *iptr will be the value of m + 1, where m is the thread index.

16) What do you expect `Grand Total` to be?

The expected value of the grand total is the sum of the numbers from 1 to 50 (inclusive), multiplied by 50,000. It can be calculated using the formula (50 * (50 + 1)) / 2 * 50,000.

17) Why you are getting different results?

The code produces different results because it suffers from a race condition. Multiple threads are modifying the shared global variable tot_items concurrently, without proper synchronization. This concurrent access can lead to unpredictable interleaving of operations, causing inconsistent and varying results. To ensure consistent results, synchronization mechanisms, such as mutex locks or atomic operations, should be used to coordinate access to the shared variable and avoid race conditions.