Name: Emad Al-Hadhrami ID:1937187

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

```
emad@lamp ~$ ./E1
Parent: My process# ---> 17741
Parent: My thread # ---> 140286092891968
Child: Hello World! It's me, process# ---> 17741
Child: Hello World! It's me, thread # ---> 140286092887808
Parent: No more child thread!
```

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

They are the same, this happens because threads within the same process share the same process ID.

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

```
emad@lamp ~$ ./E1
Parent: Global data = 5
Child: Global data was 5.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
emad@lamp ~$ ./E1
Parent: Global data = 5
Child: Global data was 10.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
```

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

No, the program may not give the same output every time it is run. The reason for this is that the execution of threads in a multi-threaded program is non-deterministic. The scheduler decides how threads are scheduled and executed, and the order in which they run can vary between different runs of the program.

5) Do the threads have separate copies of glob_data?

No, the threads in the given program do not have separate copies of the glob_data variable.

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

```
emad@lamp ~$ ./E1
I am the parent thread
I am thread #4, My ID #139845090289408
I am thread #9, My ID #139845048325888
I am thread #7, My ID #139845065111296
I am thread #5, My ID #139845081896704
I am thread #3, My ID #139845098682112
I am thread #1, My ID #139845115467520
I am thread #2, My ID #139845107074816
I am thread #8, My ID #139845056718592
I am thread #6, My ID #139845073504000
I am thread #0, My ID #139845123860224
I am the parent thread again
```

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

No, the output lines may not come in the same order every time the program is executed. The order of output lines is determined by the scheduler, which handles the concurrent execution of threads.

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

```
emad@lamp ~$ ./E1
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 139944469784320, pid: 18079, addresses: local: 0X5C68DEDC, global: 0XFA5
F407C
Thread: 139944469784320, incremented this_is_global to: 1001
Thread: 139944478177024, pid: 18079, addresses: local: 0X5CE8EEDC, global: 0XFA5
F407C
Thread: 139944478177024, incremented this_is_global to: 1002
After threads, this_is_global = 1002

Now that the threads are done, let's call fork..
Before fork(), local_main = 17, this_is_global = 17
Parent: pid: 18079, local address: 0X2AEC19C8, global address: 0XFA5F407C
Child: pid: 18082, local address: 0X2AEC19C8, global address: 0XFA5F407C
Child: pid: 18082, set local_main to: 13; this_is_global to: 23
Parent: pid: 18079, local_main = 17, this_is_global = 17
```

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

Yes, the value of this_is_global can change after the threads have finished. Multiple threads are created and executed concurrently. These threads access and modify the shared variable this_is_global.

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

The local addresses will be different for each thread.

The global addresses will be the same for all threads.

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

In the parent process, local_main and this_is_global will not change after the child process has finished. The parent process maintains its own set of variables separate from the child process.

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

The local addresses will not be the same in each process. The global addresses will initially be the same, but after modification, they will be different for each process due to copy-on-write. The reason for these differences is that each process has its own memory space, and changes made in one process's memory do not affect the memory of another process.

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

```
emad@lamp ~$ ./E1
End of Program. Grand Total = 54182834
emad@lamp ~$ ./E1
End of Program. Grand Total = 52670561
emad@lamp ~$ ./E1
End of Program. Grand Total = 55063296
emad@lamp ~$ ./E1
End of Program. Grand Total = 45798176
```

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

Total executions = 50 * 50000 = 2500000

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

Here is the breakdown of the values of *iptr during the executions:

In the first iteration, *iptr would have been 50 since tot_items is decremented by *iptr to reach the final value.

In the second iteration, *iptr would have been 49.

In the third iteration, *iptr would have been 48.

...

In the 45798176th iteration, *iptr would have been 1 since the final decrement brings tot_items to 0.

So, the values of *iptr during the executions would have ranged from 50 to 1, with each subsequent iteration decrementing *iptr by 1 until it reaches 1.

16) What do you expect Grand Total to be?

Based on the provided code, each thread increments the tot_items variable by its respective data value 50,000 times. The data value of each thread is set to m + 1, where m is the index of the thread. Since there are 50 threads and each thread increments tot_items by its data value 50,000 times, we can calculate the expected grand total by summing up all the data values and multiplying by 50,000.

Here's the calculation:

total = (1 + 2 + 3 + ... + 50) * 50,000

= (50 * 51 / 2) * 50,000

= 1,275,000,000

17) Why you are getting different results?

The reason for the different results can be attributed to the use of multiple threads accessing and modifying the shared variable tot_items concurrently.