LAB 6

EE463 Lab

Ahmed Alsaggaf   ID - 2036616

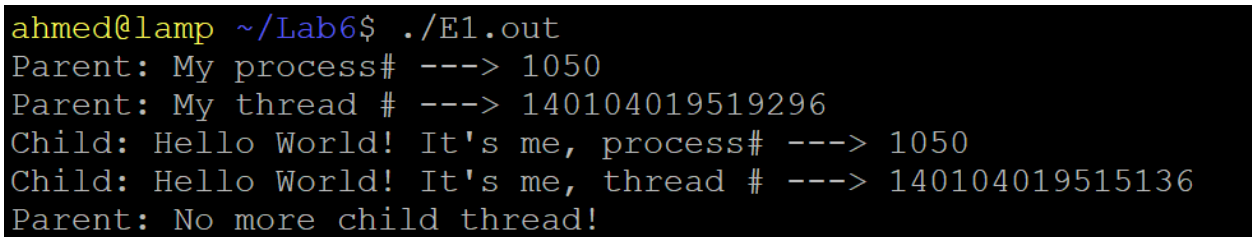
DATE:
8 - 6 - 2023

Lab Instructor:
Eng. Tukey Gari

Database:
EE-463

Section:
C3

1) Code 1:

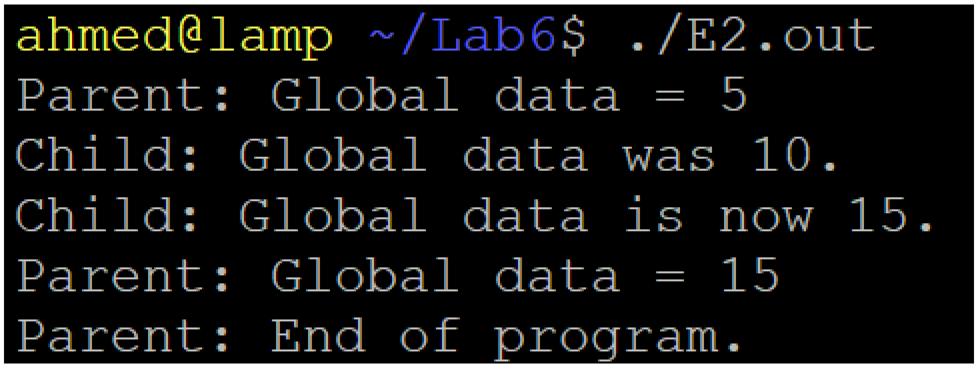


```
ahmed@lamp ~/Lab6$ ./E1.out
Parent: My process# ---> 1050
Parent: My thread # ---> 140104019519296
Child: Hello World! It's me, process# ---> 1050
Child: Hello World! It's me, thread # ---> 140104019515136
Parent: No more child thread!
```

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

Because they are part of the same process, the parent and child threads' process ID numbers are the same, but their thread IDs are different.

3) Code 2 output:



```
ahmed@lamp ~/Lab6$ ./E2.out
Parent: Global data = 5
Child: Global data was 10.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
```

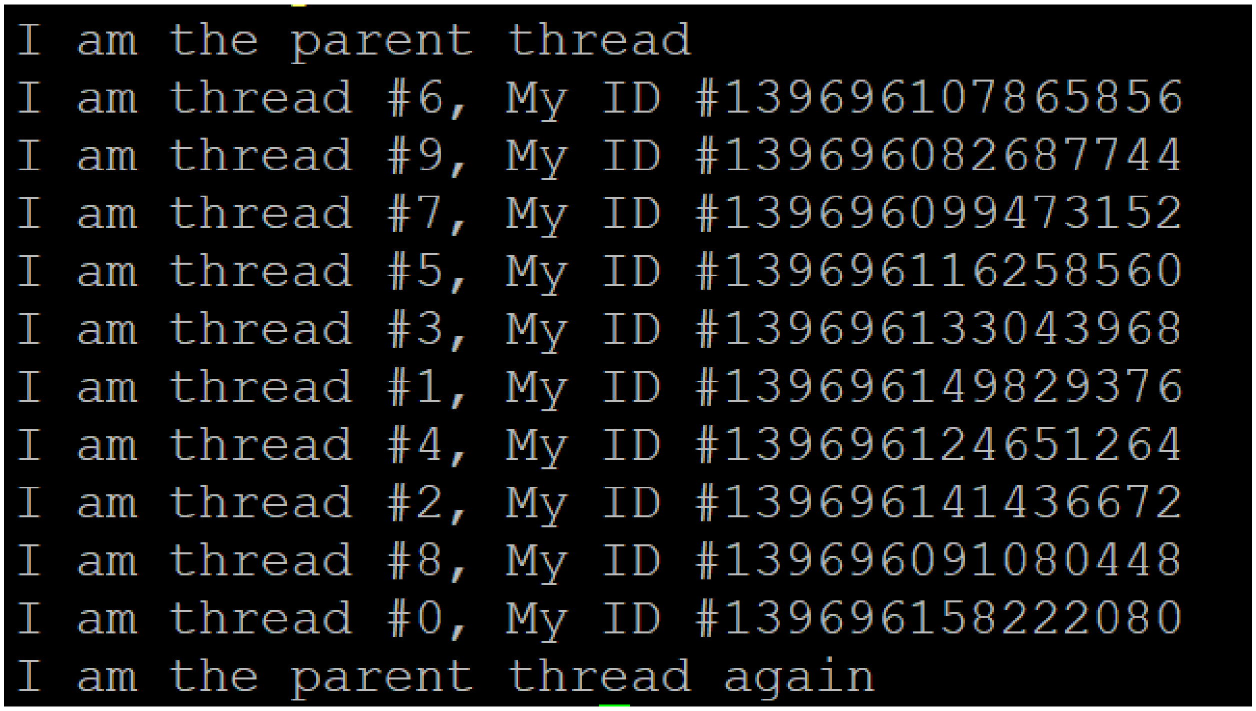
4) Does the program give the same output every time? Why?

Due to how the threads are scheduled and executed, the program may not always produce the same results. Variable thread execution timing can result in various interleavings and possibly race situations.

5) Do the threads have separate copies of glob_data?

The threads do not each have their own copy of glob_data. Since it is a global variable, all threads can access it. Every change made to glob_data by one thread is also seen by other threads.

6) Code 3 output:

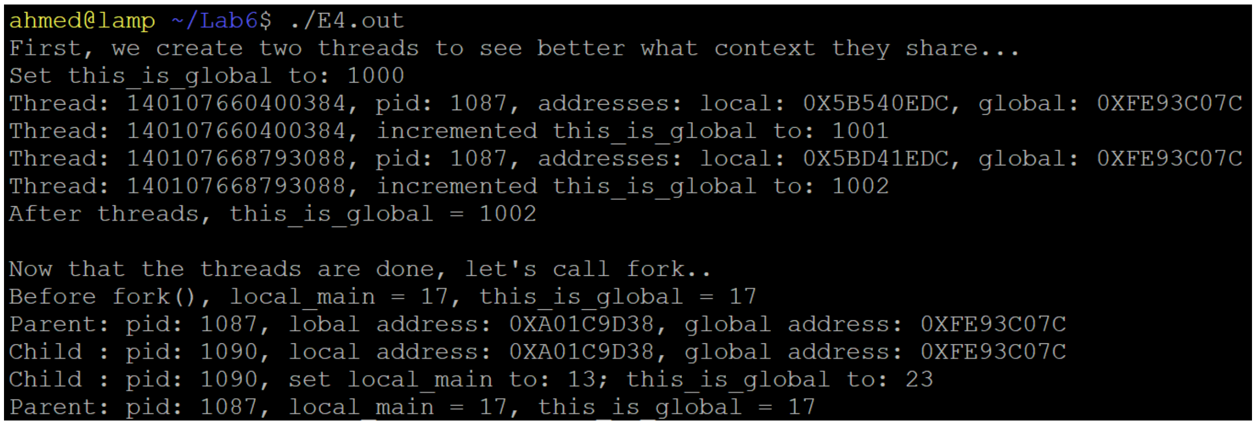


```
I am the parent thread
I am thread #6, My ID #139696107865856
I am thread #9, My ID #139696082687744
I am thread #7, My ID #139696099473152
I am thread #5, My ID #139696116258560
I am thread #3, My ID #139696133043968
I am thread #1, My ID #139696149829376
I am thread #4, My ID #139696124651264
I am thread #2, My ID #139696141436672
I am thread #8, My ID #139696091080448
I am thread #0, My ID #139696158222080
I am the parent thread again
```

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

No, the output lines might not always appear in the same order. The operating system's scheduling mechanism, the availability of CPU resources, and the timing of thread creation all have an impact on the scheduler, which decides the order in which threads are executed. Every time the program is executed, the scheduler could assign and carry out threads in a different sequence, changing the output order.

8) Code 4 output:



```
ahmed@lamp ~/Lab6$ ./E4.out
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 140107660400384, pid: 1087, addresses: local: 0X5B540EDC, global: 0XFE93C07C
Thread: 140107660400384, incremented this_is_global to: 1001
Thread: 140107668793088, pid: 1087, addresses: local: 0X5BD41EDC, global: 0XFE93C07C
Thread: 140107668793088, 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: 1087, lobal address: 0XA01C9D38, global address: 0XFE93C07C
Child : pid: 1090, local address: 0XA01C9D38, global address: 0XFE93C07C
Child : pid: 1090, set local_main to: 13; this_is_global to: 23
Parent: pid: 1087, local_main = 17, this_is_global = 17
```

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

No, when the threads have completed, the value of this_is_global did change. This_is_global was a shared global variable among the threads, so when one thread increased its value, the other threads also noticed the change.

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

In each thread, the local addresses are different. As there is a separate stack space for each thread, the local variables are kept in several memory locations. However, because each thread makes reference to the same global variable, the global address (&this_is_global) is the same across all threads.

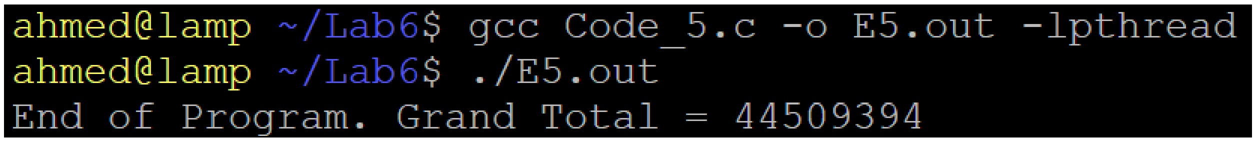
11) Did local_main and this_is_global change after the child process has finished? Why?

No, when the child process has terminated, the values of local_main and this_is_global did not change. The memory space of the parent process, including the variables' values at the moment of the fork() call, is allocated separately for the child process that fork() starts. Variables in the parent process are unaffected by changes made to those in the child process.

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

Each process uses a different set of local addresses. Fork() creates a child process that has access to its own memory area, including a stack. As a result, in comparison to the parent process, the local variables in the child process have different memory addresses.

13) Code 5 output:



```
ahmed@lamp ~/Lab6$ gcc Code_5.c -o E5.out -lpthread
ahmed@lamp ~/Lab6$ ./E5.out
End of Program. Grand Total = 44509394
```

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

Each thread executes tot_items = tot_items + *iptr many times. The value of n in the for loop within the thread_func function, which is set to 50,000 in this case, determines the precise number of executions.

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

Each thread's *iptr value during these executions will be unique. When a thread is created, its parameter, m+1, is given a unique value for each thread. Thus, for the first thread, the value of *iptr will be 1, for the second, 2, etc., up to 50 for the last thread.

16) What do you expect Grand Total to be?

Calculate the anticipated Grand Total by adding the numbers 1 through 50. It is the total of the values that were given to the threads as arguments. The projected Grand Total in this scenario would be 1 + 2 + 3 +... + 50 = 1275.

17) Why you are getting different results?

Due to a race situation, the software could provide various results. The global variable tot_items is being used by several threads at once. The order and timing of the activities may change since they are not coordinated or protected, which could produce incorrect outcomes. This race condition may lead to different tot_items final values as different threads override each other's updates.