King Abdul-Aziz University
Faculty of Engineering

Operating Systems

LAB#6

Ayad Mamdouh Aleideney	1937406

Subject & Section:

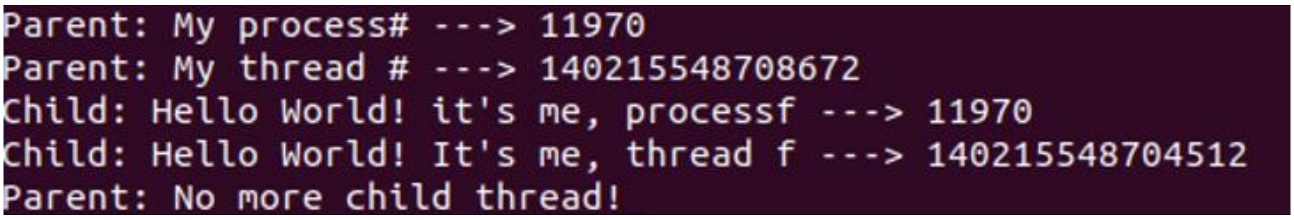
EE-463 / C3

Instructor:

Dr. Abdulghani Al-Qasmi

Q1:

1- Output:

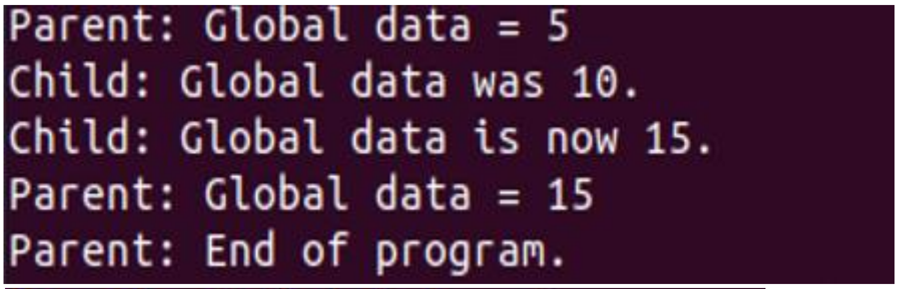


```
Parent: My process# ---> 11970
Parent: My thread # ---> 140215548708672
Child: Hello World! it's me, processf ---> 11970
Child: Hello World! It's me, thread f ---> 140215548704512
Parent: No more child thread!
```

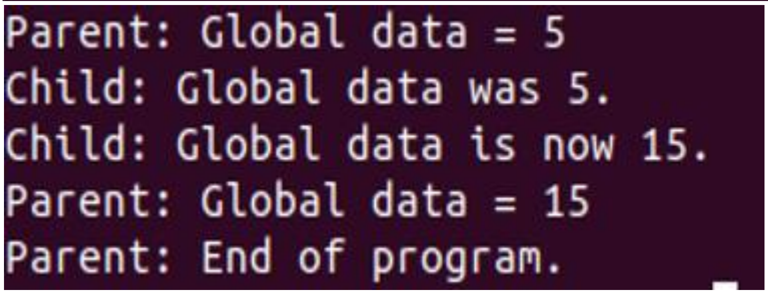
2- The parent and child threads' process IDs are identical. There is only one process, the parent process, in the example program. Using the pthread_create() function, the parent process generates a child thread. The PrintHello function, which is executed by the child thread, prints the thread ID and process ID using pthread_self() and getpid(), respectively. The process ID for both the parent and child remains the same because the child thread is created in the same process as the parent. However, because each thread has a distinct identity within the process, the thread IDs are different.

Q2:

3- Output:



```
Parent: Global data = 5
Child: Global data was 10.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
```



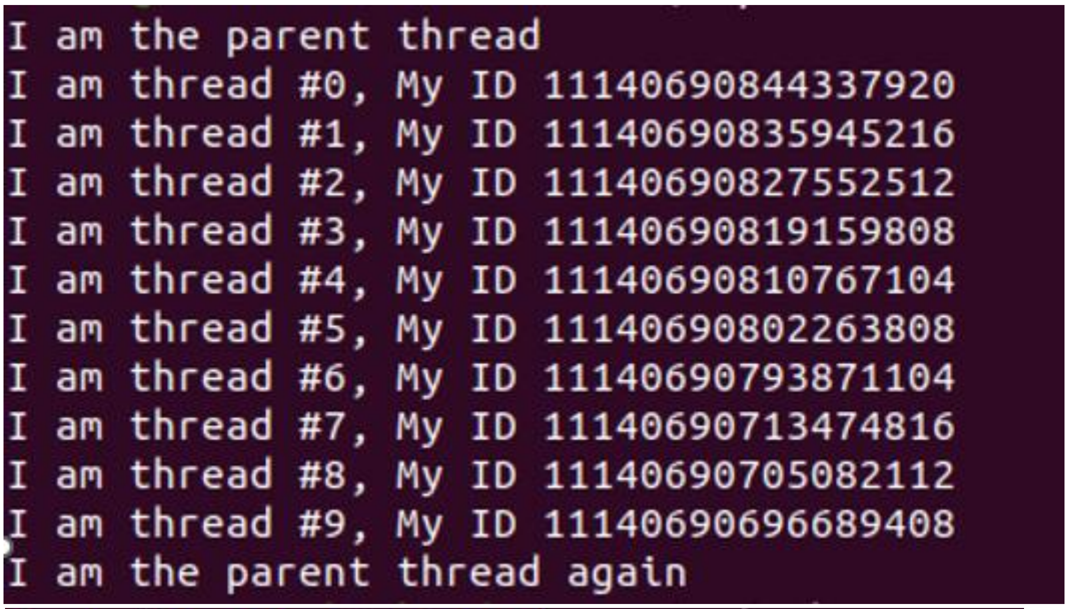
```
Parent: Global data = 5
Child: Global data was 5.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
```

4- No, the program does not always produce the same results. Because it depends on how the threads are scheduled and executed, the outcome could differ. Since both the parent thread and the child thread are active at the same time, the sequence of execution is unpredictable.

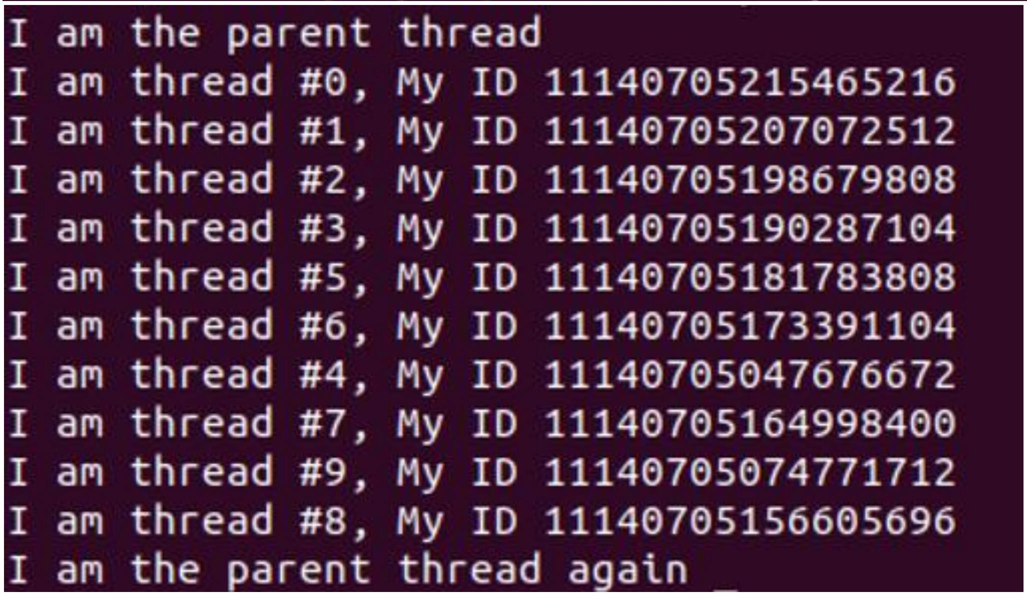
5- No, the threads do not have separate copies of glob_data. In the given example, glob_data is a shared variable used by all threads in the provided programme. The value of the variable observed by the parent thread is also changed when the child thread sets glob_data's value to 15. The memory address where glob_data is kept is shared by both threads. Other threads accessing the same variable can see any changes made by one thread.

Q3:

6- Output:



```
I am the parent thread
I am thread #0, My ID 111406908443379 20
I am thread #1, My ID 11140690835945216
I am thread #2, My ID 11140690827552512
I am thread #3, My ID 11140690819159808
I am thread #4, My ID 11140690810767104
I am thread #5, My ID 11140690802263808
I am thread #6, My ID 11140690793871104
I am thread #7, My ID 11140690713474816
I am thread #8, My ID 11140690705082112
I am thread #9, My ID 11140690696689408
I am the parent thread again
```

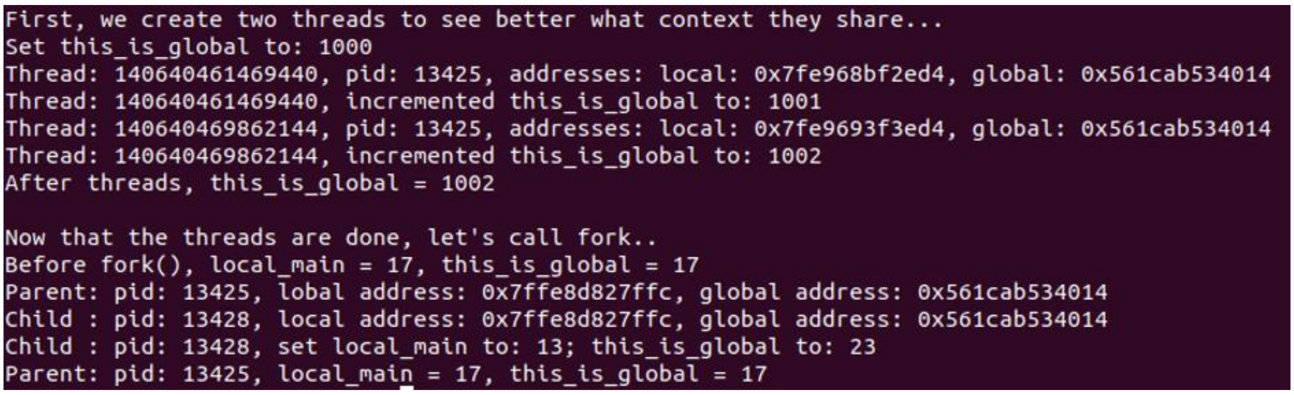


```
I am the parent thread
I am thread #0, My ID 11140705215465216
I am thread #1, My ID 11140705207072512
I am thread #2, My ID 11140705198679808
I am thread #3, My ID 11140705190287104
I am thread #5, My ID 11140705181783808
I am thread #6, My ID 11140705173391104
I am thread #4, My ID 11140705047676672
I am thread #7, My ID 11140705164998400
I am thread #9, My ID 11140705074771712
I am thread #8, My ID 11140705156605696
I am the parent thread again
```

7- No, every time the program is run, the output lines might not appear in the same order. The order in which the threads are performed is unpredictable and can change every time the program is run. The threads are individually scheduled by the operating system, and the underlying scheduling method determines the order in which they execute. As a result, the thread execution sequences may vary from run to run of the program.

Q4:

8- Output:



```
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 140640461469440, pid: 13425, addresses: local: 0x7fe968bf2ed4, global: 0x561cab534014
Thread: 140640461469440, incremented this_is_global to: 1001
Thread: 140640469862144, pid: 13425, addresses: local: 0x7fe9693f3ed4, global: 0x561cab534014
Thread: 140640469862144, 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: 13425, lobal address: 0x7ffe8d827ffc, global address: 0x561cab534014
Child : pid: 13428, local address: 0x7ffe8d827ffc, global address: 0x561cab534014
Child : pid: 13428, set local_main to: 13; this_is_global to: 23
Parent: pid: 13425, local_main = 17, this_is_global = 17
```

9- Yes, this_is_global changed after the threads have finished. It was initially set to 1000 and then incremented by both threads. The final value of this_is_global is 1002 This_is_global is a shared global variable among the threads, so any changes performed by one thread affect the others.

10- No, each thread's local addresses are not the same. The addresses of local variables vary for each thread because each has its own stack frame.

The global address (&this_is_global), on the other hand, is the same for all threads. All threads share the same global variable, which only has one instance in memory.

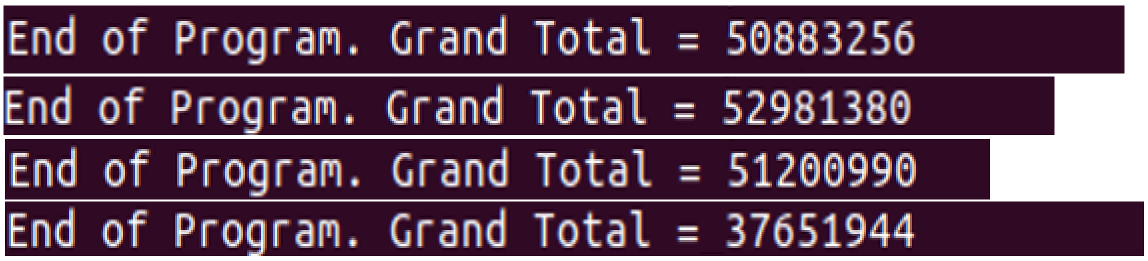
11- Local_main and this_is_global did not alter following the completion of the child process.New instances of this_is_global and local_main are created in the child process. The parent process is unaffected when these variables are changed in the child process.

12- No, not all processes use the same local addresses. The addresses of local variables vary for each process because each has its own stack. Similar to local addresses, global addresses (&this_is_global) vary depending on the process. Fork() creates a child process and gives it access to the parent's global variables as well as its own address space. The parent process is unaffected when the global variables of the child process are changed, and vice versa.

Q5:

13- When running the program multiple times, the output will vary because the execution order and timing of the threads are non-deterministic in a multi-threaded environment.

Output:



```
End of Program. Grand Total = 50883256
End of Program. Grand Total = 52981380
End of Program. Grand Total = 51200990
End of Program. Grand Total = 37651944
```

14- The line `tot_items = tot_items + (*iptr);` is executed multiple times by each thread. In this case, it is executed 50 times because there are 50 threads.

15- The value of `*iptr` during the executions will vary depending on the thread. Each thread receives a different value of `*iptr` from the `tids` array. The values range from 1 to 50 because `tids[m].data` is assigned `m + 1` in the loop.

16- The Grand Total is the sum of all the values of `*iptr` from each thread, which range from 1 to 50. The expected value of Grand Total would be the sum of integers from 1 to 50, which is 1275.

17- The concurrent execution of several threads is what causes the various results. There is a race condition created by the concurrent access and modification of the tot_items variable by the threads. The scheduling and interleaving of the threads determine the final value of tot_items. The outcome could differ every time the program is executed since the execution order and timing are not deterministic.