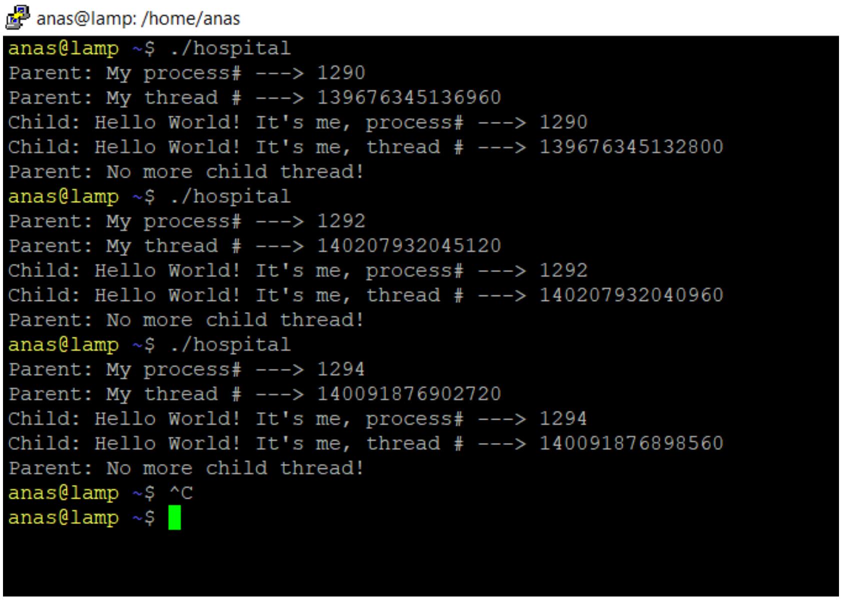
Name: Anas Hamza ID:1937572

1)

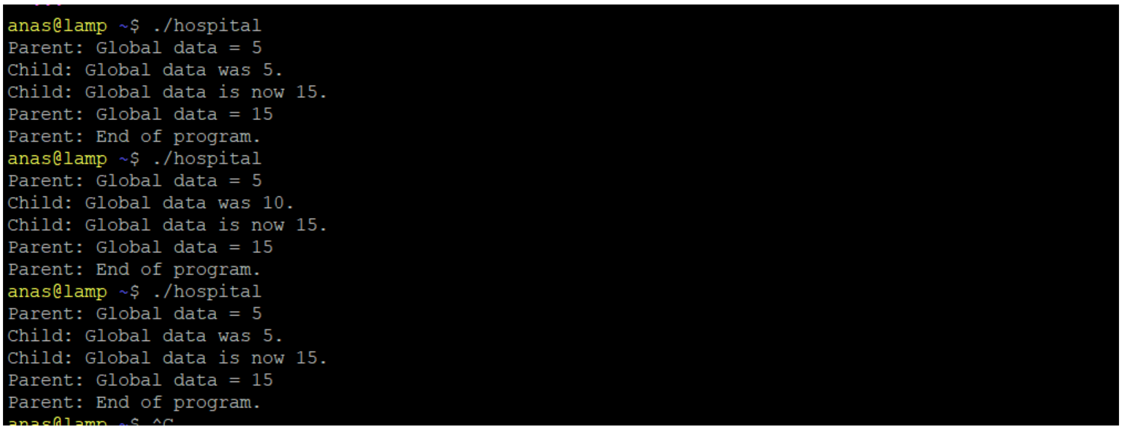


anas@lamp: /home/anas

```
anas@lamp ~$ ./hospital
Parent: My process# ---> 1290
Parent: My thread # ---> 139676345136960
Child: Hello World! It's me, process# ---> 1290
Child: Hello World! It's me, thread # ---> 139676345132800
Parent: No more child thread!
anas@lamp ~$ ./hospital
Parent: My process# ---> 1292
Parent: My thread # ---> 140207932045120
Child: Hello World! It's me, process# ---> 1292
Child: Hello World! It's me, thread # ---> 140207932040960
Parent: No more child thread!
anas@lamp ~$ ./hospital
Parent: My process# ---> 1294
Parent: My thread # ---> 140091876902720
Child: Hello World! It's me, process# ---> 1294
Child: Hello World! It's me, thread # ---> 140091876898560
Parent: No more child thread!
anas@lamp ~$ ^C
anas@lamp ~$
```

2) The process ID numbers of the parent and child threads are the same. This is because the child thread is created within the same process as the parent thread. Therefore, they both share the same process ID.

3)

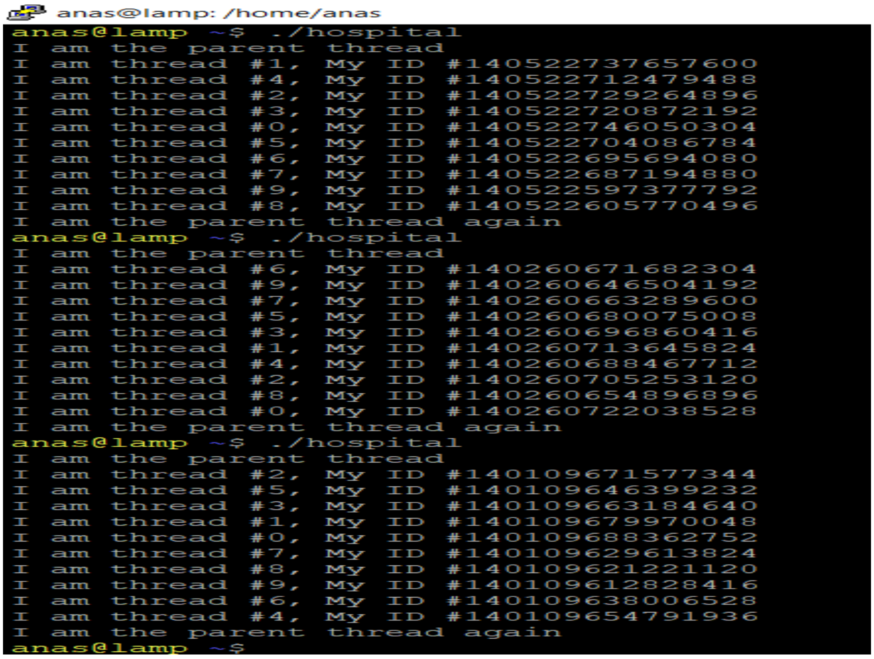


```
anas@lamp ~$ ./hospital
Parent: Global data = 5
Child: Global data was 5.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
anas@lamp ~$ ./hospital
Parent: Global data = 5
Child: Global data was 10.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
anas@lamp ~$ ./hospital
Parent: Global data = 5
Child: Global data was 5.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
anas@lamp ~$ ^C
```

4) No, because of non-deterministic thread scheduling and a lack of synchronization, the program does not provide the same output every time. The sequence in which threads are scheduled to run might change, and the lack of synchronization methods can result in race situations and unexpected behavior.

5) No, the threads do not have separate copies of the glob_data variable. The glob_data variable is a global variable, which means it is shared among all the threads in the program. Any modifications made to glob_data by one thread will be visible to all other threads. In the given code, both the parent and child threads operate on the same glob_data variable.

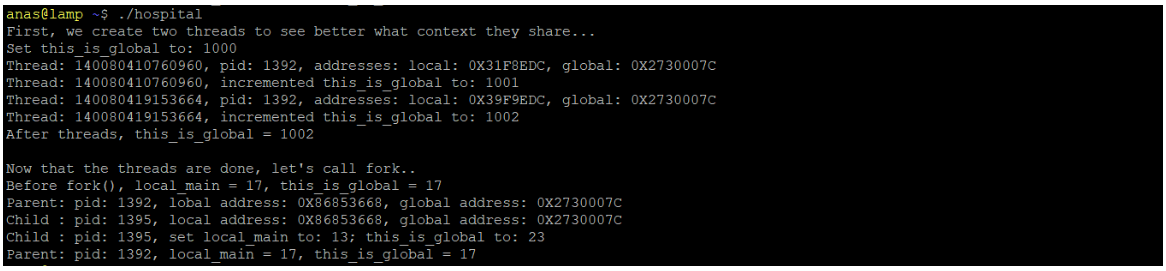
6)



```
anas@lamp ~$ ./hospital
I am the parent thread
I am thread #1, My ID #140522737657600
I am thread #4, My ID #140522712479488
I am thread #2, My ID #140522729264896
I am thread #3, My ID #140522720872192
I am thread #0, My ID #140522746050304
I am thread #5, My ID #140522704086784
I am thread #6, My ID #140522695694080
I am thread #7, My ID #140522687194880
I am thread #9, My ID #140522597377792
I am thread #8, My ID #140522605770496
I am the parent thread again
anas@lamp ~$ ./hospital
I am the parent thread
I am thread #6, My ID #140260671682304
I am thread #9, My ID #140260646504192
I am thread #7, My ID #140260663289600
I am thread #5, My ID #140260680075008
I am thread #3, My ID #140260696860416
I am thread #1, My ID #140260713645824
I am thread #4, My ID #140260688467712
I am thread #2, My ID #140260705253120
I am thread #8, My ID #140260654896896
I am thread #0, My ID #140260722038528
I am the parent thread again
anas@lamp ~$ ./hospital
I am the parent thread
I am thread #2, My ID #140109671577344
I am thread #5, My ID #140109646399232
I am thread #3, My ID #140109663184640
I am thread #1, My ID #140109679970048
I am thread #0, My ID #140109688362752
I am thread #7, My ID #140109629613824
I am thread #8, My ID #140109621221120
I am thread #9, My ID #140109612828416
I am thread #6, My ID #140109638006528
I am thread #4, My ID #140109654791936
I am the parent thread again
anas@lamp ~$
```

7) The output lines do not come in the same order every time because the threads are executing concurrently, and their scheduling and execution order is determined by the operating system's thread scheduler. The scheduling algorithm, system workload, and other factors can result in different thread execution orders, leading to variations in the output order.

8)



```
anas@lamp ~$ ./hospital
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 140080410760960, pid: 1392, addresses: local: 0X31F8EDC, global: 0X2730007C
Thread: 140080410760960, incremented this_is_global to: 1001
Thread: 140080419153664, pid: 1392, addresses: local: 0X39F9EDC, global: 0X2730007C
Thread: 140080419153664, 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: 1392, lobal address: 0X86853668, global address: 0X2730007C
Child : pid: 1395, local address: 0X86853668, global address: 0X2730007C
Child : pid: 1395, set local_main to: 13; this_is_global to: 23
Parent: pid: 1392, local_main = 17, this_is_global = 17
```

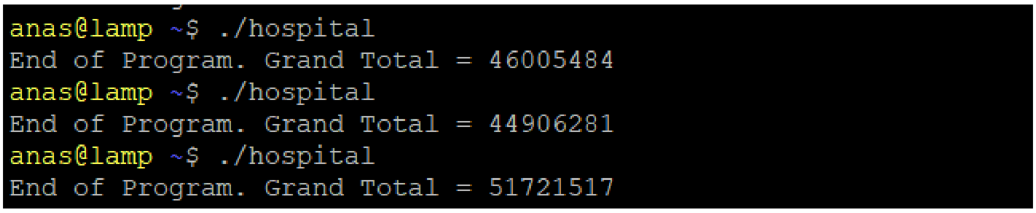
9) Yes, the value of this_is_global changed after the threads have finished. Initially, this_is_global was set to 1000 before creating the threads. Then, each thread incremented its value by 1, resulting in this_is_global becoming 1002. This change is due to the fact that threads share the same memory space within a process. When one thread modifies a shared variable, the change is visible to other threads as well. Therefore, both threads were able to access and modify the same this_is_global variable, leading to its value.

10) No, the local addresses are not the same in each thread. Each thread has its own stack memory, and local variables are stored on the stack. Therefore, the addresses of local variables will be different for each thread.On the other hand, the global address remains the same for all threads. Global variables are stored in a shared data segment, and all threads can access and modify them using the same address.

11) No, the values of local_main and this_is_global did not change after the child process finished. The child process has its own separate memory space, so the modifications it made to those variables do not affect the variables in the parent process.

12) The local addresses are different in each process, while the global addresses are the same. Each process has its own stack memory, so the local addresses will be different. However, global variables are stored in a shared data segment, resulting in the same global addresses for both processes.

13)



```
anas@lamp ~$ ./hospital
End of Program. Grand Total = 46005484
anas@lamp ~$ ./hospital
End of Program. Grand Total = 44906281
anas@lamp ~$ ./hospital
End of Program. Grand Total = 51721517
```

14) The given program exhibits a race condition where multiple threads concurrently access and modify the shared variable tot_items without synchronization. This leads to unpredictable results as the final value of tot_items depends on the interleaving of thread executions. To ensure consistent and correct results, synchronization mechanisms such as locks or mutexes should be used to coordinate access to the shared variable.

15) The line tot_items = tot_items + *iptr; is executed 50,000 times within a loop in the thread_func function. This loop is invoked by each of the 50 threads created in the main function. Therefore, the line is executed a total of 2,500,000 times in the program.

16) To calculate the expected "Grand Total," (50* (50 + 1)) / 2 *50000=63750000

17) The different results observed in each run of the program are due to a race condition caused by concurrent access and modification of the shared variable tot_items without proper synchronization. The non-deterministic order of thread execution and inconsistent updates to tot_items lead to varying results. Synchronization mechanisms are necessary to ensure consistent and predictable outcomes in multi-threaded programs.