NAME : hatem ali aloufi

ID: 1937396

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

My Output:

```
Parent: My process# ---> 3380

Parent: My thread # ---> 140117725988672

Child: Hello World! It's me, process# ---> 3380

Child: Hello World! It's me, thread # ---> 140117723313728

Parent: No more child thread!
```

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

The process ID (PID) numbers of the parent and child threads are the same. Due to the fact that thread create creates new threads within the same process as the parent thread, this is the result. As a result, the process ID is shared by the parent and child threads.

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

1.

```
Parent: Global data = 5

Child: Global data was 10.

Child: Global data is now 15.

Parent: Global data = 15

Parent: End of program.
```

2.

```
Parent: Global data = 5

Child: Global data was 5.

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, Because the order of execution and thread scheduling is non-deterministic. Sometimes the parent thread may execute its operations first and print a value of 10, followed by the child thread printing the previous value of 10 and then updating it to 15. In this case, the parent thread will print a final value of 15.

5) Do the threads have separate copies of glob data?

No, the threads in the program do not have separate copies of glob data. The variable glob data is a global variable, and global variables are shared among all threads within the same process.

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

```
I am the parent thread

I am thread #4, My ID #3043994480

I am thread #5, My ID #3035601776

I am thread #6, My ID #3027209072

I am thread #3, My ID #3052387184

I am thread #2, My ID #3060779888

I am thread #7, My ID #3018816368

I am thread #8, My ID #3010423664

I am thread #9, My ID #3002030960

I am thread #1, My ID #3069172592

I am thread #0, My ID #3077565296

I am the parent thread again
```

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

No, the output lines do not come in the same order every time. The order of the output depends on how the computer schedules and runs the threads. The operating system decides when each thread gets to run based on factors like priorities and other activities happening on the computer. As a result, even though the threads are created and joined in a particular order, the order in which they actually print their messages can vary from run to run.

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

```
First, we create two threads to see better what context they share...

Set this_is_global to: 1000

Thread: 3070053232, pid: 10487, addresses: local: 0XB6FD438C, global: 0X804A040

Thread: 3070053232, incremented this_is global to: 1001

Thread: 3078445936, pid: 10487, addresses: local: 0XB77D538C, global: 0X804A040

Thread: 3078445936, 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: 10487, lobal address: 0XBFF5CB0C, global address: 0X804A040

Child : pid: 10490, local address: 0XBFF5CB0C, global address: 0X804A040

Child : pid: 10490, set local main to: 13; this_is_global to: 23

Parent: pid: 10487, local main = 17, this_is_global = 17
```

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

Yes, this_is_global changed after the threads have finished. In the program, the threads are incrementing the value of this_is_global by 1. Since the threads share the same memory space, they access and modify the same variable. As a result, each thread increments this_is_global, leading to a change in its value.

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

No, the local addresses are not the same in each thread. Local variables in each thread have a unique memory address because each thread has its own stack. Each thread's local variables (local_thread) have a unique address in the output. On the other hand, each thread uses the same global addresses. All threads have access to a shared memory area where global variables are kept. As a result, each thread uses the same address for the global variable (this_is_global).

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

No changes were made to local_main or this_is_global after the child process was finished. The fork() function copies the parent process, down to the values of its variables, when a child process is created. Any adjustments made to the variables in the child process, however, have no impact on the parent process. The parent process' values are left unaltered while the child process makes changes to its own copies of local_main and this_is_global in the programme.

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

No, the local addresses are not the same in each process. Each process has its own separate memory space, including its own stack. Therefore, the local variables (local_main) have different memory addresses in each process. In the output, the addresses of local_main are different for the parent and child processes.

Similarly, the global addresses are the same for the parent and child processes. The same global variables are shared by the parent and child processes in memory. Both the parent and child processes share the same address for the global variable (this_is_global) in the output.

The child process is created as a copy of the parent process after the fork() function is called. The child process, on the other hand, has a distinct memory area, so any modifications made to variables in one process do not affect the other. As a result, while the addresses of global variables remain the same, those of local variables change.

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

My Outputs:

End of Program. Grand Total = 38245310

End of Program. Grand Total = 33871115

End of Program. Grand Total = 18193003

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. As each thread runs a loop 50,000 times, so the line is executed 50,000 times in each thread.

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

The value of *iptr represents the data passed to each thread. In this program, *iptr is a pointer to the integer value stored in the data member of the tidrec structure. Each thread is passed a different data value, ranging from 1 to 50. Therefore, during the executions, *iptr takes on values from 1 to 50, depending on the specific thread.

16) What do you expect Grand Total to be?

The expected Grand Total depends on the number of threads (NTIDS) and the data values passed to each thread. In this program, the Grand Total is calculated by summing the values of *iptr in each thread and updating the shared global variable tot items. Since *iptr ranges from 1 to 50 and there are 50 threads, the expected Grand Total would be the sum of numbers from 1 to 50, which is 1,275.

17) Why are you getting different results?

I think its because each time you run the program due to the race condition caused by multiple threads modifying the shared variable tot items simultaneously without synchronization. When multiple threads try to update tot items concurrently, conflicts occur as they read and write the variable simultaneously. The final result becomes unpredictable because the threads' operations can overlap and interfere with each other.

18) Write your answers to all the above questions with evidence of your experiments and submit them in one .pdf file.

Done.