King Abdulaziz University
Faculty of Engineering
Electrical & Computer Eng
Department

LAB #6

Operating Systems— EE463

Lecturer(s): Dr. Abdulghani M. Al-Qasimi
& Eng. Turki Abdul Hafeez
3rd Semester Spring 2023

	Name	ID
1	Khalid Al-mutairi	1743292

2- The reason why the process ID numbers of the parent and child threads are the same is because they both belong to the same process. In a multi-threaded process, all threads share the process resources, including memory and file descriptors. This shared nature of resources is the reason why they have identical process IDs. However, it's important to note that each thread has its own distinct thread identifier, as they represent individual threads executing within the same process.

3- I run the program several times. The output varied in each run due to the non-deterministic nature of thread scheduling.

4- The program does not produce consistent output each time it is run due to the unpredictable nature of multithreading. The concurrent execution of threads introduces non-determinism, as the order in which threads are scheduled and executed is determined by the operating system's thread scheduler. The shared global variable **glob_data** accessed and modified by both the parent and child threads creates a race condition. The final output of the program is influenced by factors such as the timing of thread execution, the occurrence of context switches between threads, and the scheduling decisions made by the operating system. These factors contribute to the variability in the program's output.

5- No, threads in a multithreaded program do not have individual copies of the **glob_data** variable. Instead, they share the same global memory space. This means that all threads have direct access to the global variables, including **glob_data**. Any changes made to **glob_data** by one thread are immediately visible to all other threads within the program. The shared nature of global memory allows threads to communicate and manipulate shared data efficiently.

6- Upon running the program multiple times, I noticed that the output differs on each run. The variability in output arises from the non-deterministic execution order of the threads. The program's behavior is influenced by the scheduling decisions made by the operating system, leading to different orders in which the threads execute. Consequently, the output lines corresponding to the threads appear in varying sequences in each run.

7- No, the order of output lines is not consistent in every run of the program. The reason behind this lies in the non-deterministic nature of thread execution. The specific order in which threads are scheduled and executed is determined by the operating system's thread scheduler. This scheduling decision takes into account various factors, including system load, thread priorities, and other dynamic factors. Therefore, when multiple threads are created, such as in this program with a loop, there is no guarantee about the precise order in which they will be scheduled and executed by the operating system. As a result, the output lines can appear in different orders with each run of the program.

8- Indeed, the global variable **this_is_global** did undergo modification once the threads completed their execution. Within the program, each thread incremented the value of **this_is_global** by some value 'l'. This behavior arises from the fact that threads within the same process share a common memory space. Consequently, any modifications made by one thread to a global variable are immediately observable to all other threads operating within the same process. Thus, when one thread modifies the global variable **this_is_global**, the updated value becomes visible to all other threads.

9- The local addresses differ in each thread since each thread possesses its own stack space. Consequently, the address of the local variable **local_thread** varies across threads. However, the global addresses remain consistent across threads. This consistency stems from the shared global memory among all threads. As a result, the address of the global variable

this_is_global remains the same for every thread since they all access the same global memory space.

10- No, the variables **local_main** and **this_is_global** in the parent process did not change after the child process completed its execution. This behavior is due to the nature of process creation using the **fork** system call. When a new process is forked from the parent process, it receives a separate copy of the entire memory of the parent process. As a result, any modifications made by the child process to its own copies of **local_main** and **this_is_global** variables do not impact the values of these variables in the parent process. The parent and child processes operate with their distinct memory spaces, ensuring that changes in one process do not affect the variables in the other process.

11- The local and global addresses appear to be the same in each process at the moment of the fork operation. This behavior is a result of how fork operates in Unix-based systems. When fork is invoked, the operating system generates a nearly identical replica of the existing process, including its memory layout. However, it is crucial to understand that these memory layouts are separate copies. Any modifications made in one process's memory do not impact the memory of the other process. Although the memory addresses may appear identical, they represent distinct and isolated memory regions in both the child and parent processes.

12- The line tot_items = tot_items + *iptr; is executed 50,000 times by each thread. As there are 50 threads, it should execute a total of 500.000 times.

13- *iptr has values ranging from I to 50, because it points to tids[m].data in main, which is assigned the value m+l.

14- The expected "Grand Total" is the sum of the series I to 50 (i.e., I + 2 + 3 + ... + 50), multiplied by 50,000, because each of these numbers is added to tot items 50,000 times. So the answer is close to $3,187,500,000$.

15- The issue arises due to a phenomenon called a race condition, which is a common challenge in concurrent programming. It occurs when multiple threads attempt to access and modify shared data simultaneously. In this scenario, the variable **tot_items** is shared among all threads, and each thread aims to increment it without any synchronization mechanism in place.