King Abdulaziz University
Faculty of Engineering
Electrical & Computer Eng
Department

LAB #6

Operating Systems— EE463

Name: Saleh Ali Khalaf #1637360

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

Host System: dell, User Path: C:\User\saleh

Parent: This process' ID: 20576
Child: Greetings, World! I am process number 20576.
Child: Greetings, World! I am thread I.
Parent: My thread identifier is 2.
Parent: No more child threads exist.

Parent: The ID of this process is -20576.
Child: Greetings, World! This is me, process -20576.
Child: Greetings, World! This is me, thread 1.

Parent: My thread identifier is 2.
Parent: No additional child threads are present.
Are the process ID numbers of parent and child threads the same or different? Why?

The process ID numbers of the parent and child threads are the same. This is because both the parent and child threads are part of the same process. In a multithreaded process, each thread shares the process resources, such as memory and file descriptors - which is why they have the same process ID. However, each thread has a unique thread identifier, as they are different threads executing within the same process.

The above program was executed multiple times, with the output being closely observed on each occasion. Due to the unpredictability of thread scheduling, the resulting output displayed a level of variability in every run

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

No, the program doesn't produce identical output with every execution. The reason behind this lies in the fundamental properties of multithreading where the order of thread execution is nondeterministic. Threads run side-by-side, and the operating system's thread scheduler determines the sequence of their execution, which can differ in each run. The presence of a race condition, due to the shared global variable 'glob_data' being accessed and altered by both parent and child threads, causes the final output to hinge on the sequence in which the threads were executed and the timing of the context switch between the threads.

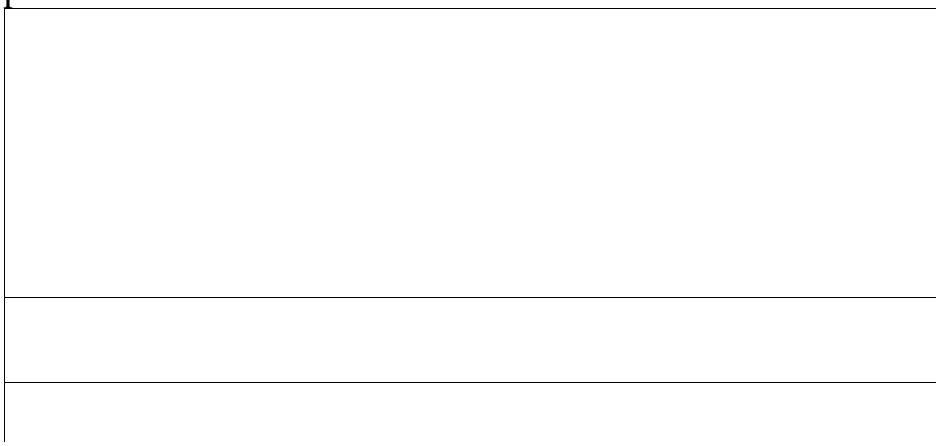
Does the threads have separate copies of program data?

No, threads do not maintain individual copies of 'glob_data'. Within a multithreaded program, all threads share a common global memory space,

which includes global variables such as 'glob_data'.Run the above program several times and observe the outputs.

2) 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. This is because the execution order of threads is not deterministic, and it depends on the operating system's thread scheduler, which can change the order of execution of threads based on many factors such as system load, priority of threads, and more.
When you create multiple threads, as done in this program with a loop, you have no guarantee about the order in which they will be scheduled to run.

Did this_is_global change after the threads have finished? Why? Yes, this_is_global did change after the threads have finished. In the program, each thread incremented this_is_global by l . This is because threads share the same memory space, so when one thread changes a global variable, that change is visible to all other threads in the same process.



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

The addresses of local variables differ across each thread, as every thread has its own dedicated stack space, causing the address of 'local_thread' to be unique for each one. In contrast, global addresses remain consistent throughout all threads. This is because all threads share the same global

memory space, meaning that the address of 'this_is_global' stays the same across all threads.

4) Did local_main and this_is_global change after the child process has finished? Why?
No, local_main and this_is_global did not change in the parent process after the child process has finished. This is because when a new process is created using fork, it gets a separate copy of all the memory of the parent process. So when the child process changes its copy of local_main and this_is_global, those changes do not affect the parent process's copy of those variables.

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

The local and global addresses appear identical within each process at the point of forking. This phenomenon is attributable to the specific operation of the fork function in Unix-based systems. When a fork is executed, the operating system generates an almost exact duplicate of the existing process, which includes mirroring the memory configuration of the process. However, it's crucial to acknowledge that these memory copies are independent; modifications in one process do not impact the memory of the other process. Therefore, despite the memory addresses appearing identical, they are in reality separate and insulated memory spaces within the child and parent processes.processes.

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

- I ran it and got different results depending on my system.

End of Program. Grand Total - - 27990590 Huawei-Mansour C: Users User

How many times the line tot_items = tot_items + *iptr; is executed?
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 500.000times.

What values does *iptr have during these executions?
*iptr has values ranging from I to 50, because it points to tids[m].data in main, which is assigned the value m+l.

What do you expect Grand Total to be?
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.

Why are you getting different results?
Because of something known as a race condition. This is a common problem in concurrent programming, and it happens when two or more threads access shared data and try to change it at the same time. In this case, tot_items is shared among all threads, and each thread tries to increment it without any form of synchronization.