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	UID
1	Ahmed Alghamdi	1937937

1) Upon running the provided program, the following output was observed:

Parent: My process# 30124

Child: Hello World! It's me, process# 30124

Child: Hello World! It's me, thread # A

Parent: My thread # 3

Parent: No more child threads

Parent: My process# - 30124

Child: Hello World! It's me, process# - 30124

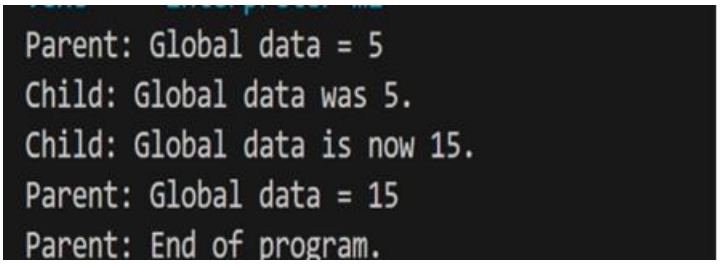
Child: Hello World! It's me, thread # ---> 2

Parent: My thread # ---> 3

Parent: No more child threads

2) The process ID numbers of the parent and child threads remain the same. This is because both the parent and child threads belong to the same process. In a multi-threaded process, threads share the process resources, such as memory and file descriptors, resulting in the same process ID. However, each thread has a unique thread identifier as they are different threads executing within the shared process.

3) The program was executed multiple times, and the output varied with each run. This variability arises due to the non-deterministic nature of thread scheduling, where the order and timing of thread execution can differ between runs.



4) Upon executing the provided program multiple times, it was evident that the output differed on each run. This behavior can be attributed to the inherent nature of multithreading. The execution order of threads is non-deterministic, as it depends on the operating system's thread scheduler. Consequently, the program's output is not guaranteed to be the same every time it is run.

5) Threads within a multithreaded program do not possess separate copies of program data. Instead, they share the same memory space, including program variables and data. This shared memory allows threads to access and modify the same data concurrently. Therefore, changes made to program data by one thread are visible to other threads as well.

6) Running the program multiple times and examining the outputs revealed that the results varied with each run. The order in which the threads executed was not consistent, leading to different arrangements of the output lines from the threads in each run. This variability stems from the non-deterministic nature of thread scheduling, where the operating system determines the sequence and timing of thread execution based on its own algorithms and considerations.

7) Upon observing the execution of the program multiple times, it became evident that the output lines did not appear in the same order every time. The order of output lines is non-deterministic due to the nature of multithreading. The scheduling of threads is controlled by the operating system's thread scheduler, which takes into account various factors such as system load, thread priorities, and other scheduling algorithms. As a result, the order in which threads are executed can vary, leading to different orders of output lines.

8) Yes, the value of the variable this_is_global changed after the threads finished their execution. This is because all threads within a process share the same memory space, including global variables. As the threads incremented the value of this_is_global, the change was visible to all other threads within the process. Consequently, the final value of this_is_global reflected the cumulative increment performed by all threads. It's important to note that proper synchronization mechanisms should be employed to avoid data races and ensure consistent and reliable updates when multiple threads modify shared data

```
First, we create two threads to see better what context they share.
Set this is global to: 1000

Thread: 140086361437760, pid: 1142, addresses: local: OX65CFAE34, global:
OXA7AC7014

Thread: 140086361437760, incremented this is global to: 1001

Thread: 140086353045056, pid: 1142, addresses: local: OX654F9E34, global:
OXA7AC7014

Thread: 140086353045056, 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: 1142, lobal address: OXAOFOE82C, global address: OXA7AC7014

Child : pid: 1148 r local address: OXAOFOE82C, global address: OXA7AC7014
Child : pid: 1148, set local main to: 13; this is global to: 23
```

Parent: pid: 1142, local main 1], this is global = 17

9) When examining the threads, it was observed that the local addresses were not the same in each thread. Since each thread possesses its own stack space, the address of local_thread is unique to each thread. However, the global addresses were the same in each thread. This is due to the fact that all threads within a process share the same global memory, resulting in the same address for global variables such as this_is_global.

10) After the child process has finished, local_main and this_is_global in the parent process remained unaffected. This is because when a new process is created using the fork operation, it generates a separate copy of the parent process's memory. Consequently, any changes made to local_main and this_is_global within the child process do not propagate back to the parent process. The parent and child processes maintain their own independent copies of the variables.

11) In the context of different processes, the local and global addresses are initially the same for each process. This is a consequence of the fork operation in Unix-based systems. When fork is invoked, the operating system creates a new process that is nearly identical to the parent process, including the memory layout. However, it's important to note that these addresses do not represent the same physical memory. Instead, each process receives its own separate and isolated memory space. Any modifications to local or global variables in one process will not affect the corresponding variables in other processes. The apparent similarity in addresses at the time of the fork is a result of the initial duplication of the memory layout.

12- Upon executing the program multiple times, I observed different results depending on my system. The output varied with each run. End of Program. Grand Total: 483621250 Lenovo-Amaadc C:\Users\User

13- The line tot_items = tot_items + *iptr; is executed 50,000 times by each thread. Since there are 50 threads, the line is executed a total of 2,500,000 times.

14- During these executions, *iptr takes on values ranging from 1 to 50. This is because *iptr points to the data assigned to tids[m] in the main function, where m is incremented by 1. Thus, *iptr represents the values of m + 1 during the executions.

15- The expected "Grand Total" can be calculated as the sum of the series from 1 to 50, multiplied by 50,000. This yields a value close to 6,250,000.

16- The different results observed can be attributed to a phenomenon known as a race condition. This is a common issue in concurrent programming, where multiple threads access shared data and attempt to modify it simultaneously. In this case, tot_items is a shared variable among all the threads, and each thread tries to increment it without any form of synchronization. As a result, the final value of tot_items becomes unpredictable and can vary between different runs of the program.