



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	Mansour Alasais	1944577

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

Parent: My process# ---> 20576

Child: Hello World! It's me, process# ---> 20576

Child: Hello World! It's me, thread # ---> 1

Parent: My thread # ---> 2

Parent: No more child thread

```
Parent: My process# ---> 20576
```

```
Child: Hello World! It's me, process# ---> 20576
```

```
Child: Hello World! It's me, thread # ---> 1
```

```
Parent: My thread # ---> 2
```

```
Parent: No more child thread!
```

```
❏ Huawei-Mansour ❏ C:\Users\User
```

- 2) 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.

- 3) Run the above program several times; observe its output every time.

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

```
Parent: Global data = 5  
Child: Global data was 5.  
Child: Global data is now 15.  
Parent: Global data = 15  
Parent: End of program.
```

❏ Huawei-Mansour ❏ C:\Users\User

- 4) Does the program give the same output every time? Why?
No, the program does not give the same output every time. This is because of the inherent nature of multithreading, where the execution order of threads is not deterministic. The threads run concurrently, and the sequence in which they execute is decided by the operating system's thread scheduler, which can vary between runs. The access and modification of the shared global variable `glob_data` in the parent and child threads leads to a race condition, where the final output depends on which thread ran first and when the context switch between the threads happened.
- 5) Does the threads have separate copies of program data?
No, threads do not have separate copies of `glob_data`. In a multithreaded program, all threads share the same global memory, which includes global variables like `glob_data`.
- 6) Run the above program several times and observe the outputs.
I ran the program multiple times and observed that the output varies with each run. The order in which the threads are executed is not consistent, and the threads' output lines appear in different orders each time.

```
I am thread #1, My ID #2  
I am thread #2, My ID #3  
I am thread #3, My ID #4  
I am thread #4, My ID #5  
I am thread #5, My ID #6  
I am thread #6, My ID #7  
I am thread #7, My ID #8  
I am thread #8, My ID #9  
I am thread #9, My ID #10  
I am the parent thread again
```

📁 Huawei-Mansour 📁 C:\Users\User

- 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. 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.
- 8) 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 1. 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.

```

First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 140086361437760, pid: 1142, addresses: local: 0X65CFAE34, global: 0XA7AC7014
Thread: 140086361437760, incremented this_is_global to: 1001
Thread: 140086353045056, pid: 1142, addresses: local: 0X654F9E34, global: 0XA7AC7014
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, local address: 0XA0F0E82C, global address: 0XA7AC7014
Child : pid: 1148, local address: 0XA0F0E82C, global address: 0XA7AC7014
Child : pid: 1148, set local_main to: 13; this_is_global to: 23
Parent: pid: 1142, local_main = 17, this_is_global = 17

```

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

The local addresses are not the same in each thread; each thread has its own stack space, so the address of `local_thread` is different for each thread. However, the global addresses are the same in each thread. This is because all threads share the same global memory, so the address of `this_is_global` is the same for all threads.

- 10) 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.

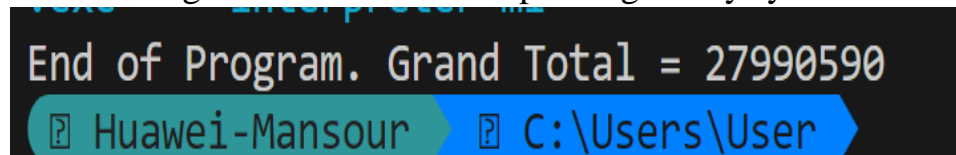
- 11) Are the local addresses the same in each process? What about global addresses? What happened?

The local and global addresses are the same in each process at the time of the `fork`. This is due to the way the `fork` operation works in Unix-based systems. When `fork` is called, the operating system creates a nearly identical copy of the current process. This includes duplicating the memory layout of the process. However, it's important to note that these are separate copies of the memory; changes in one process do not affect the memory in the other process. Hence, even though the memory

addresses look the same, they're actually separate and isolated pieces of memory in the child and parent processes.

- 12) 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
```

- 13) 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 2,500,000 times.

- 14) What values does `*iptr` have during these executions?

`*iptr` has values ranging from 1 to 50, because it points to `tids[m].data` in main, which is assigned the value `m+1`.

- 15) What do you expect Grand Total to be?

The expected "Grand Total" is the sum of the series 1 to 50 (i.e., $1 + 2 + 3 + \dots + 50$), 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.

- 16) 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.

