



Faculty of Engineering and Technology
Department of Electrical and Computer Engineering
ENCS3390
Operating Systems
THE FIRST PROGRAMMING TASK

Prepared by: Lana Mahmoud Ali Musaffer

ID: 1210455

Instructor: Dr. Bashar Tahayna

Section: 4

PART #1: Process Management

For the process management, I implemented a matrix multiplication using multiple child processes. First, I initialize matrices “ID” and “myOwnMatrix” and fill them based on my ID number and birth year. Then, I divide the matrix multiplication task among multiple child processes, so that each child process calculates a part of the resulting matrix in the specified range of rows; to allow parallel execution for better performance. Finally, I implemented **void cleanup()** function for closing the read and write ends of pipes.

In the main, I begin by creating pipes for communication between parent and child processes. Then, forks child processes to perform the task concurrently. After the child processes perform their computations, the parent process collects the results and prints the final matrix, which is named “multMatrix”. Note that the parent process waits for all child processes to complete before printing the result, this ensures that the child processes have finished their computations.

In general, In this approach each process run in their own address space, providing a high degree of **isolation**. So, if one process fails, it does not directly affect other processes. Processes can run concurrently on multi-core systems, allowing for **parallelism**, this lead to better utilization of resources and improved performance.

PART #2 + PART #4: Multithreaded Processing & Threads Management

For the multithreaded processing part, I did the same task, but here I divide the task among multiple threads.

For the joinable threads, I arranged the data that each thread should work in a struct, then I created a pool of threads; for better control over thread creation and destruction, so I can manage concurrency in an efficient way. The implementation of **ThreadFunc()** is similar to **childProcess()**, this function performs matrix multiplication but within a thread, so that each thread computes a specific range of rows in the result matrix. Also here, cleanup are taking into count.

The detached threads implementation is the same as the joinable, but there is differences. In joinable threads, the main program explicitly waits for joinable threads to finish using the **pthread_join()** function, and it require explicit cleanup by the main program using. However, in detached threads, the main program does not need to wait for their completion, and resources associated with detached threads are automatically released when they complete, and the system takes care of the cleanup. Moreover, in joinable threads, the main program waits until a joinable thread completes its execution, for ensuring synchronization between the main program and the thread. In the other hand, in detached threads, the main program can continue its execution without waiting for detached threads to finish.

For the mixed threading approach, I implement the matrix task by making half of the threads detached **pthread_detach()**, and the other half joined **pthread_join()**. This solution increase **parallelism**, detached threads can run independently, providing parallelism that may improve the overall performance of the application. It also provides **resource Cleanup**, detached threads automatically release resources upon completion, reducing the burden on the main program to manage thread resources explicitly. Joinable threads allow the main program to wait for specific threads, providing **control over synchronization** when needed. In addition, the ability to choose between detached and joinable threads provides **flexibility** in designing the threading model based on the specific requirements of different tasks.

PART #3: Performance Measurements

I compared the performance of each approach (native, process, and threads) by measuring the time that it takes to perform the matrix multiplication. This involves recording the time that taken from beginning to end of the task (Wall Time = CPU time + OS + I/O overhead) , using **gettimeofday()** function.

➤ *First, I measure and compare the outcome of all solutions in seconds:*

	FIRST RUN	SECOND RUN	THIRD RUN	FOURTH RUN	FIFTH RUN
NAIVE APPROACH	0.003638	0.003634	0.002422	0.003359	0.003617
2 CHILD PROCESSES	0.005980	0.007382	0.005416	0.006017	0.004824
2 JOINABLE THREADS	0.002512	0.002883	0.002316	0.002358	0.002159
2 DETACHED THREADS	0.000136	0.000162	0.000124	0.000142	0.000145
2 MIXED THREADS	0.002210	0.002692	0.002283	0.002519	0.002404

➤ *Second, I try to vary the number of processes and threads:*

	FIRST RUN	SECOND RUN	THIRD RUN	FOURTH RUN	FIFTH RUN
2 CHILD PROCESSES	0.004883	0.005968	0.006016	0.004976	0.004144
4 CHILD PROCESSES	0.002861	0.003485	0.003145	0.002878	0.002739
5 CHILD PROCESSES	0.001895	0.002119	0.002203	0.002588	0.002179
10 CHILD PROCESSES	0.002475	0.002488	0.002451	0.002449	0.002523
20 CHILD PROCESSES	0.002751	0.002735	0.002796	0.002810	0.002739
50 CHILD PROCESSES	0.006869	0.006926	0.006901	0.007049	0.006067
2 JOINABLE THREADS	0.001987	0.002763	0.002435	0.001935	0.002144
4 JOINABLE THREADS	0.001516	0.001845	0.001398	0.001609	0.001488
10 JOINABLE THREADS	0.002239	0.002346	0.001763	0.002098	0.002216
20 JOINABLE THREADS	0.002371	0.002337	0.002279	0.002627	0.002659
50 JOINABLE THREADS	0.003053	0.002697	0.002974	0.003137	0.002762
2 DETACHED THREADS	0.000025	0.000030	0.000042	0.000028	0.000037
4 DETACHED THREADS	0.000054	0.000049	0.000050	0.000040	0.000043
10 DETACHED THREADS	0.000133	0.000194	0.000157	0.000149	0.000160
20 DETACHED THREADS	0.000309	0.000320	0.000318	0.000330	0.000342
50 DETACHED THREADS	0.000878	0.001054	0.000957	0.001058	0.000980
2 MIXED THREADS	0.002498	0.001997	0.002296	0.001877	0.001789
4 MIXED THREADS	0.001543	0.001676	0.001452	0.001720	0.001567
10 MIXED THREADS	0.002169	0.002365	0.001830	0.002308	0.002319
20 MIXED THREADS	0.002424	0.002539	0.002449	0.002440	0.002567
50 MIXED THREADS	0.002344	0.002583	0.002658	0.002506	0.002740

✚ Note that the “naive approach” is implemented without using any child processes or threads.

✚ **FINAL RESULT:** Threads > process > naive approach

❖ ANALYSIS:

In the first look it seems that the naive approach is better than the process-based solution, but when I increased the number of child process in ranges from 3 to 5 for example, the time measured became less than the time in the naïve approach, which is make sense. Also, when I tried to increase the number of child processes let’s say more than 10, it will take more time than the naïve approach. These results are expected due to many factors such as, process creation, synchronizing, complexity, and context switching between processes is generally slower(than both approaches the naïve and threads).

Threads approach are the best solution comparing to child process and naïve approaches, it depends on many reasons. Communication between processes are through IPC can make it more challenging than sharing data between threads, as threads typically have separate memory spaces. Threads can synchronize their activities through like mutex, barrier, and other mechanisms, allowing for more fine-grained control over shared resources. Note that the mixed threads results are similar to the joinable threads due to the combination of joinable and detached threads.

Detached threads measure the time just in the main without the measuring functions time, so the time is not accurate because I used **gettimeofday()**. After that, I used global counter for making sure that each thread is done its execution, to measure the actual time. PS: the submitted code is in the second method, and the first is commented. The below table shows the result when I used the **global counter**:

	FIRST RUN	SECOND RUN	THIRD RUN
2 DETACHED THREADS	0.006222	0.006582	0.006766
4 DETACHED THREADS	0.006495	0.007650	0.007602
10 DETACHED THREADS	0.023469	0.019006	0.021003
20 DETACHED THREADS	0.045029	0.043028	0.041905
50 DETACHED THREADS	0.044775	0.044036	0.047518

Now, for the optimal number of child processes/threads, it is obviously seen through experimentation and performance testing from the previous table (the long one) that the proper number is 4-to-5 (approximately). In addition, I think selecting the number of child process or threads is related to the number of cores in my CPU, I work on i5-7200u 7th generation pc which offers two CPU cores that work with up to 4 threads at once.

➤ **Run with the best cases:**

- ✓ NUM_OF_THREADS = 4
- ✓ NUM_OF_CHILDREN = 5
- ✓ DETACHED WITH GLOBEL COUNTER TIME MEASUREMENT.

```
*****
TIME MEASURED FOR NAIVE APPROACH: 0.003635 seconds.
*****
*****
*****
*****
TIME MEASURED FOR PROCESS SOLUTION: 0.002903 seconds.
*****
*****
*****
*****
TIME MEASURED FOR JOINABLE THREADS: 0.001553 seconds.
*****
*****
*****
*****
TIME MEASURED FOR DETACHED THREADS: 0.007554 seconds.
*****
*****
*****
*****
TIME MEASURED FOR MIXED THREADS: 0.001470 seconds.
*****
```

CHALLENGES

✚ I faced ~~Race condition~~ issues while implementing threads, and solve it using **Mutex**, this achieved thread synchronization by allowing threads to lock and unlock access to shared resources, and it prevents multiple threads from accessing critical sections concurrently, avoiding data corruption and race conditions.

—> **REFERENCE:** <https://www.youtube.com/watch?v=oq29KUy29iQ&t=319s>

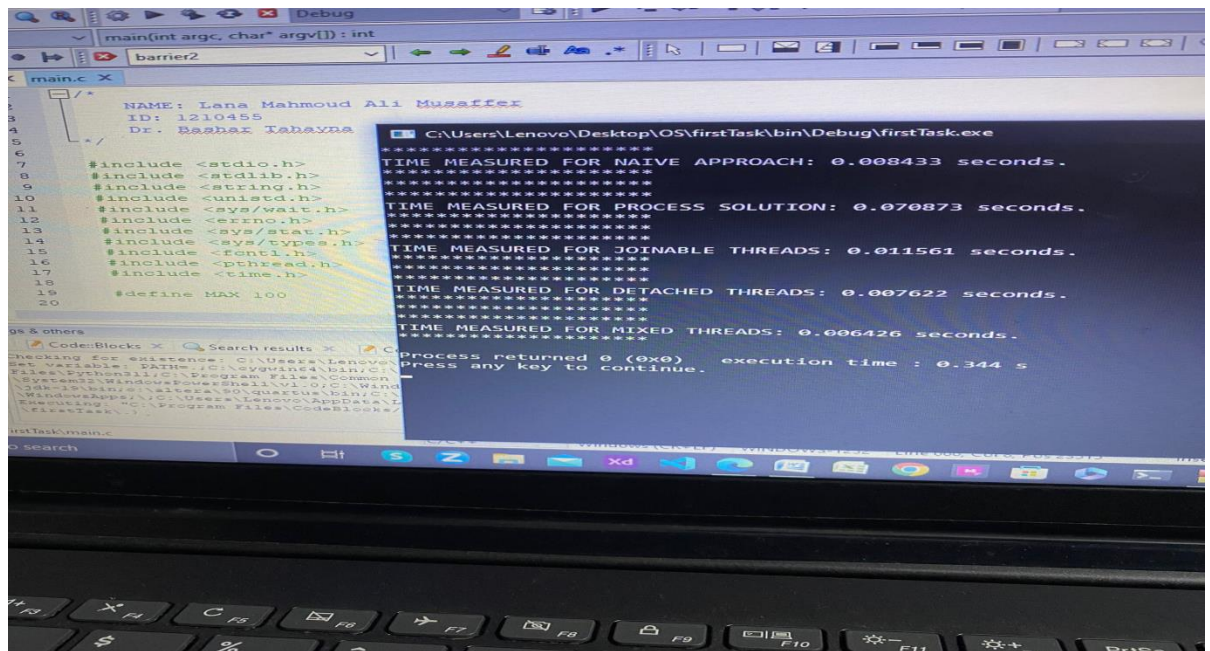
✚ Instead of Mutex, I implied **Memory Barrier** solution for achieving thread synchronization and to solve race conditions.

--> **PS:** Dr. Bashar told us that we can use it and not to use mutex in the last lecture.

✚ I submitted two C codes, the one named **“firstTask”** is the final edited clean code. **“finalTest”** code is **the same** but there are a lot of comments, **they are not useless**. I didn't remove them because they shows different approaches I try to attempt and used them here in my analysis, such as the wrong previous detached time calculation (using **gettimeofday()** -- > I commented it after solve it by using global counter), printing the matrices, and mutex.

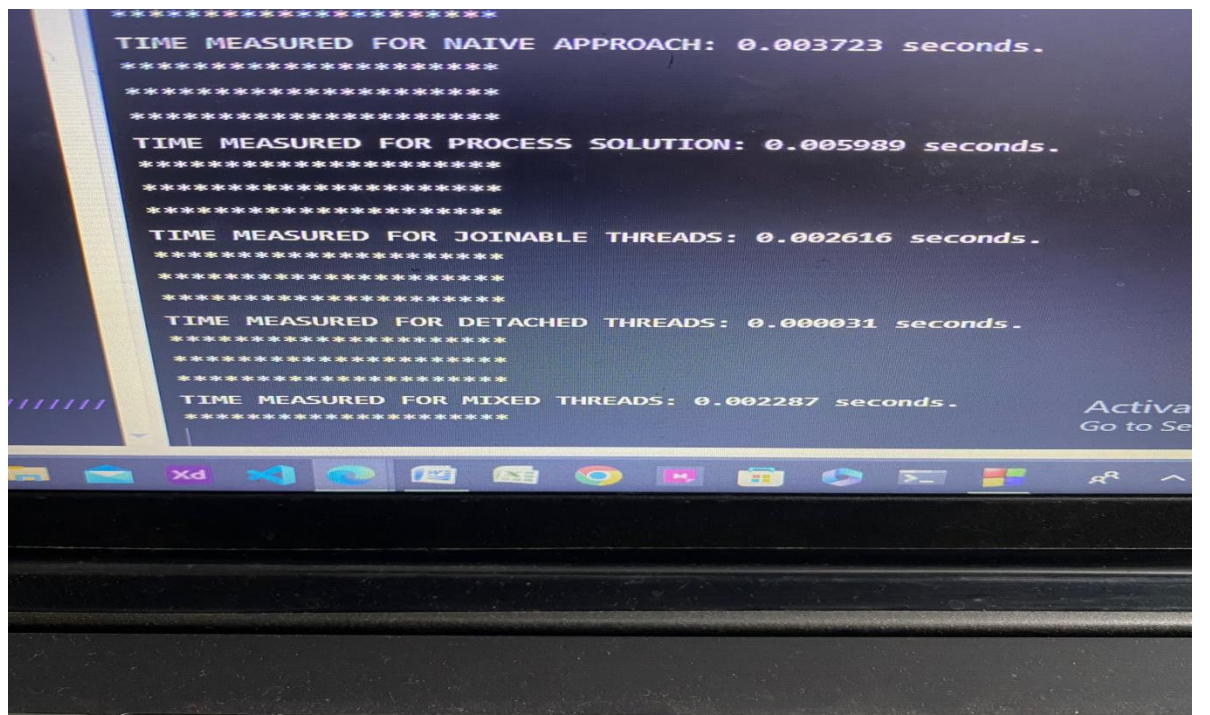
✚ I downloaded a compiler tool called **“Cygwin”** for a Linux-like experience on my laptop that works in Windows operating system, and worked in Code Blocks IDE. In the end I noticed that it affected my results, it caused context switching issues and race conditions when trying varying the number of threads. So, in my report I discussed the results that I got when running my code on **C online compiler**, in the below pictures you can notice the **HUGE difference**.

➤ In Cygwin compiler:



```
main(int argc, char* argv[]) : int
NAME: Lena Mahmoud Ali Musaffek
Dr. Basheer Tahayna
C:\Users\Lanovo\Desktop\OS\firstTask\bin\Debug\firstTask.exe
*****
TIME MEASURED FOR NAIVE APPROACH: 0.008433 seconds.
*****
TIME MEASURED FOR PROCESS SOLUTION: 0.070873 seconds.
*****
TIME MEASURED FOR JOINABLE THREADS: 0.011561 seconds.
*****
TIME MEASURED FOR DETACHED THREADS: 0.007622 seconds.
*****
TIME MEASURED FOR MIXED THREADS: 0.006426 seconds.
*****
Process returned 0 (0x0)   execution time : 0.344 s
Press any key to continue.
```

➤ In C online compiler:



```
*****
TIME MEASURED FOR NAIVE APPROACH: 0.003723 seconds.
*****
*****
TIME MEASURED FOR PROCESS SOLUTION: 0.005989 seconds.
*****
*****
TIME MEASURED FOR JOINABLE THREADS: 0.002616 seconds.
*****
*****
TIME MEASURED FOR DETACHED THREADS: 0.000031 seconds.
*****
*****
TIME MEASURED FOR MIXED THREADS: 0.002287 seconds.
*****
*****
```