## BCIT

## Comp 4956 System Programming

## Technical Programming Option

## Option Head Mirela Gutica

## Fall 2022

## Mark: \_\_\_\_\_\_\_\_ /100

**Name(s): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Lab 1

Solve the following exercises:

**Part 1**

You can find the Process Explorer and the two .exe files (MultiThreads.exe and Multithreads\_CSharp.exe) under Lab 1.

1. (50p) Use the Process Explorer in Windows to observe the execution of the two multithreaded processes (MultiThreads.exe written in C and Multithreads\_CSharp.exe written in C#). Note that I created threads in the user space that are mapped on kernel-level threads for both solutions. Each process has a main thread, a counting thread and counting threads. You can find the Process Explorer and the two .exe files under Lab 1. **Answer the following questions and include screen shots to justify your answers**:
   1. Start Process Explorer.
      1. In Process Explorer double click on the process name to open a window that presents process’ execution parameters.

Graphical user interface, text

Description automatically generated

* + 1. Run the processes few times.
  1. How many threads are created for the C process? For example, if you create 3 counting threads, is the number of threads created for this process 5 or more? Are there additional threads? Why?
     1. **Answer:** Starting the C process with 3 counting threads, there is a count of 4 total threads, per the Process Explorer. It looks as though there is a thread being used for the base application. And then 3 more threads are created for the 3 counting threads. 1 specifically keeps track of the time while the other 2 threads count. These all run within the application process.

Table

Description automatically generated with medium confidence

* 1. How many threads are created for the C# process? For example, if you create 3 counting threads, is the number of threads created for this process 5 or more? Are there additional threads? Why?
     1. **Answer:** Starting the C# process with 3 counting threads, there is a count of 14 total threads, per the Process Explorer. Again, 3 more threads are created for the 3 counting threads. 1 specifically keeps track of the time while the other 2 threads count. However, there are also independent callback threads, and threads which perform other purposes. From the class discussion on Tuesday, it seems that the ntdlls are created by the OS. There also seems to be a CLR thread independent of counting executions.

Graphical user interface, text

Description automatically generated

* 1. In which process (C or C#) threads count to a larger number? Why?
     1. **Answer:** The C process counting threads count to a larger number since there is less context switching. C# process’s context switching occurs more often, which interrupts the counting process since it takes time to cancel the thread and start it again. It is computationally intensive, which in turn, reduces the C# counting thread count.
  2. If you run the C and the C# processes several times and create the same number of counting threads (e.g., 3), are the number of threads created for each process the same?
     1. **Answer:** Yes and no. Regarding no, sometimes I see threads in the C and C# processes with the start address of an independent callback or some other start addresses not related to the counting threads created. These may start and drop throughout the processes’ life. Regarding yes, it seems a consistent number of threads are created based upon the selection of threads of the program. When 3 counting threads are selected in the programs, 3 threads seem to be created for job.
  3. Do threads in the same process end at the same count? Why?
     1. **Answer:** No, but the C# process produces very consistent results of counts between the counting threads, usually a 1 or 2 count difference between the counting threads. The C process has way more variation between the counts of the counting threads but counts to a much larger number.



Shape

Description automatically generated

* 1. What kind process executes more context switches? Why?
     1. **Answer:** I believe the C# process executes more context switches since it is constantly switching between the counting threads and outputs a smaller count numbers. The count output is also smaller since context switching is expensive. The C# process does more context switching because
  2. Identify the time spent in user mode and in kernel mode. Why do processes spend time in kernel mode?
     1. **Answer:** The user mode handles the interaction with GUI. Kernel mode manages that I/O with actual functionality and system call from user mode. Kernel mode also handles the graphics rendering of the app. Processes spend time in kernel modes because the process (its threads) accesses the CPU for computation. That is also where managers and schedulers manage the process for the CPU to execute. These processes must be done in kernel mode because it must access the hardware and lower systems of the machine.
  3. How much faster is the multithreading execution of the process in C compared to the one in C#?
     1. **Answer:** Many times faster. Creates a lot more counts.
  4. Do the following experiment: run both processes with 2 counting threads. Then run both processes with 4 counting threads.
     1. Is the performance in execution twice faster for C for 2 threads?

Graphical user interface

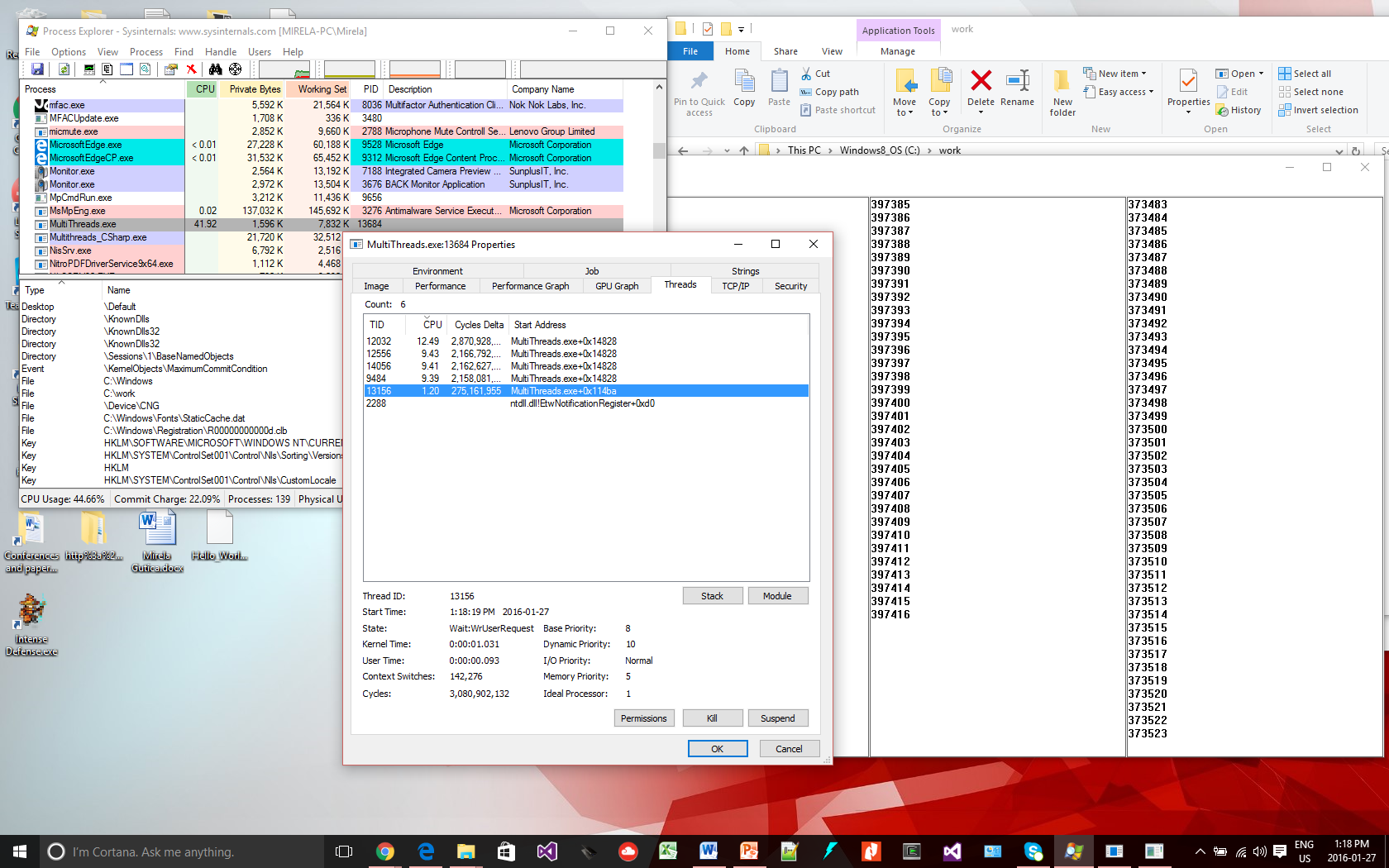
Description automatically generated

* + - 1. The C process is more than twice as fast as C#.
    1. Is the performance in execution twice faster for C# for 4 threads?

Graphical user interface, application, table

Description automatically generated

* + - 1. The C process is still more than twice as faster than the C# process.
  1. Consider the class example **MultiThreads\_SharedVariable\_CSharp.exe:**
     1. Understand the code.
     2. Run the application several times and observe in Process Explorer the threads that are created.
     3. Remove the mutex from the code. Run the application for several number of threads and max increment value for each thread. Is the shared variable incremented correctly? Why or why not?
        1. No, because the shared variable is not locked and the threads access it at unpredictable times.



**Part 2**

1. Read the documentation for: CreateThread() and CreateProcess() (C/C++) from: <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread> and <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa>
2. Read the documentation for CreateThread in C#: <https://support.microsoft.com/en-ca/help/815804/how-to-create-a-thread-by-using-visual-c> and <https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?redirectedfrom=MSDN&view=netframework-4.8>
3. (10p) Write two paragraphs that summarize the items 2 and 3. **Use your own words!**
   1. Links from item 2 describe creating a thread within the virtual address space of the calling process to execute. This is done with the CreateThread() function. The number of threads which can be created for a process is restricted base upon the available virtual memory space. The best approach for performance is to create 1 thread per processor and querying with the context information. The CreateProcess() function creates a new process with a primary thread for it. The process runs in the security context of the calling process. The process is given a process identifier. The thread created with the process is also assigned a thread identifier. The identifiers can be used until the process terminates.
   2. To use multithreading in apps using the .NET framework one must use the namespace: System.Threading. With the namespace in use, you create a Thread type and initialize a thread to it. The thread class also has a method in which to start the thread for the process and other methods and properties to change its behavior.
4. (40p) Based on your readings, the examples on MSDN and the class example **MultiThreads\_SharedVariable\_CSharp.exe**, implement a Windows application (in C#) that:
   1. has several threads and one or more shared variables
   2. each thread modifies the shared variable(s)
   3. you display the shared variable on the screen on a GUI of your choice
   4. you synchronize the access to the variable as we discussed in class

**Please note:**

Very important if you work in C#:

Use the System.Threading namespace: <https://docs.microsoft.com/en-us/dotnet/api/system.threading?view=netframework-4.8>

If you use .NET, you will notice that the .NET Framework includes classes that implement Windows synchronization objects: events, mutexes, semaphores, timers, and other Windows synchronization objects. They are available through managed wrappers found in the System.Threading namespace. The “lock” of a shared variable is implemented with a mutex (<https://docs.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=netframework-4.8>). Please note the WaitOne() and ReleaseMutex() methods.

There is an issue in C# of accessing GUI components (i.e. text boxes) from multiple threads. The issue is related to the fact that Windows forms are not thread-safe: a control that is modified from multiple threads is not guaranteed to be modified correctly. In order to access controls correctly, one solution is to use delegates. Use the Invoke method of a control with a delegate. See examples at:

<http://msdn.microsoft.com/en-us/library/ms171728.aspx>