WESTMINSTER INTERNATIONAL UNIVERSITY IN TASHKENT

COURSEWORK SUBMISSION FORM

STUDENT USE		STAFF USE	
Module Name	Concurrent Programming	First Marker's (acts as signature)	
Module Code	6BUIS011C-n	Second Marker's (acts as signature)	
Lecturer Name	Mohamed Uvaze Ahamed Ayoobkhan	Agreed Mark	
UoW Student IDs		**For Registrar's office use only (hard copy submission)**	
WIUT Student IDs	00010881		
Deadline date	March 31, 2022		
Assignment Type	□Group□Individual		

SUBMISSION INSTRUCTIONS

COURSEWORKS *must* be submitted in *both* HARD COPY (to the Registrar's Office) *and* ELECTRONIC unless instructed otherwise.
For hardcopy submission instructions refer to:
http://intranet.wiut.uz/Shared%20Documents/Forms/AllItems.aspx - Coursework hard copy submission instructions.doc
For online submission instructions refer to:
http://intranet.wiut.uz/Shared%20Documents/Forms/AllItems.aspx - Coursework online submission instructions.doc

MARKERS FEEDBACK (Continued on the next page)

Instructions to run the application

Change the connection string in App.config file of ClientSide winforms project so your directory directs to the SwipesRecord.mdf file.

Application development

The application is responsible for collecting clock-ins and clock-outs (swipes) from several terminals and saves the information into the database. Besides that, it allows only 3 terminals to make swipes at the same time. Application has been developed using .NET technologies such as Windows Forms, Windows Communication Foundation service, ADO.NET and Visual studio IDE. Also, Microsoft SQL Server with SQL Server Management Studio have been used to develop the database.

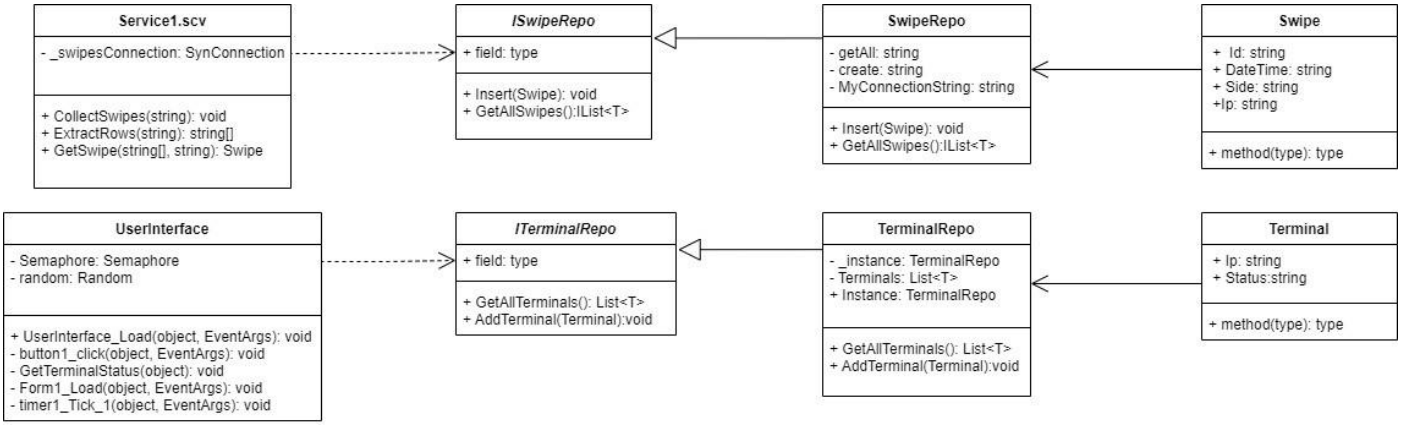
The Solution contains two projects, one for the User interface which is Windows forms application and WCF Service which is responsible for Collecting data about swipes and saving it into the database, make sure to limit the number of terminals it is getting the data from to 3 at one time, contains the local database with “mdf” extension and it also references the SynConnectDLL class library which is responsible for validating the IP address of the terminals and generates swipes that will contain Id, the datetime of when the swipe has happened and the direction. Windows forms in turn Contains data set that displays data from the database on DataGrid view on the main page.

In order to make sure that only 3 terminals could make swipes the Semaphore has been used. Here is an example of the usage of this variable:

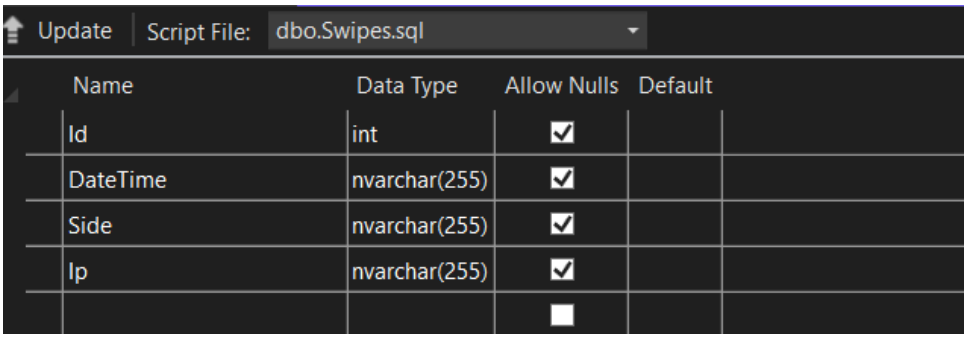
```
private static readonly Semaphore Semaphore = new Semaphore(3, 3);
```

The first attribute of the instantiation is the initial number of entries, and the second attribute is for the maximum number of requests for semaphore that can be granted concurrently.

UML Diagram:

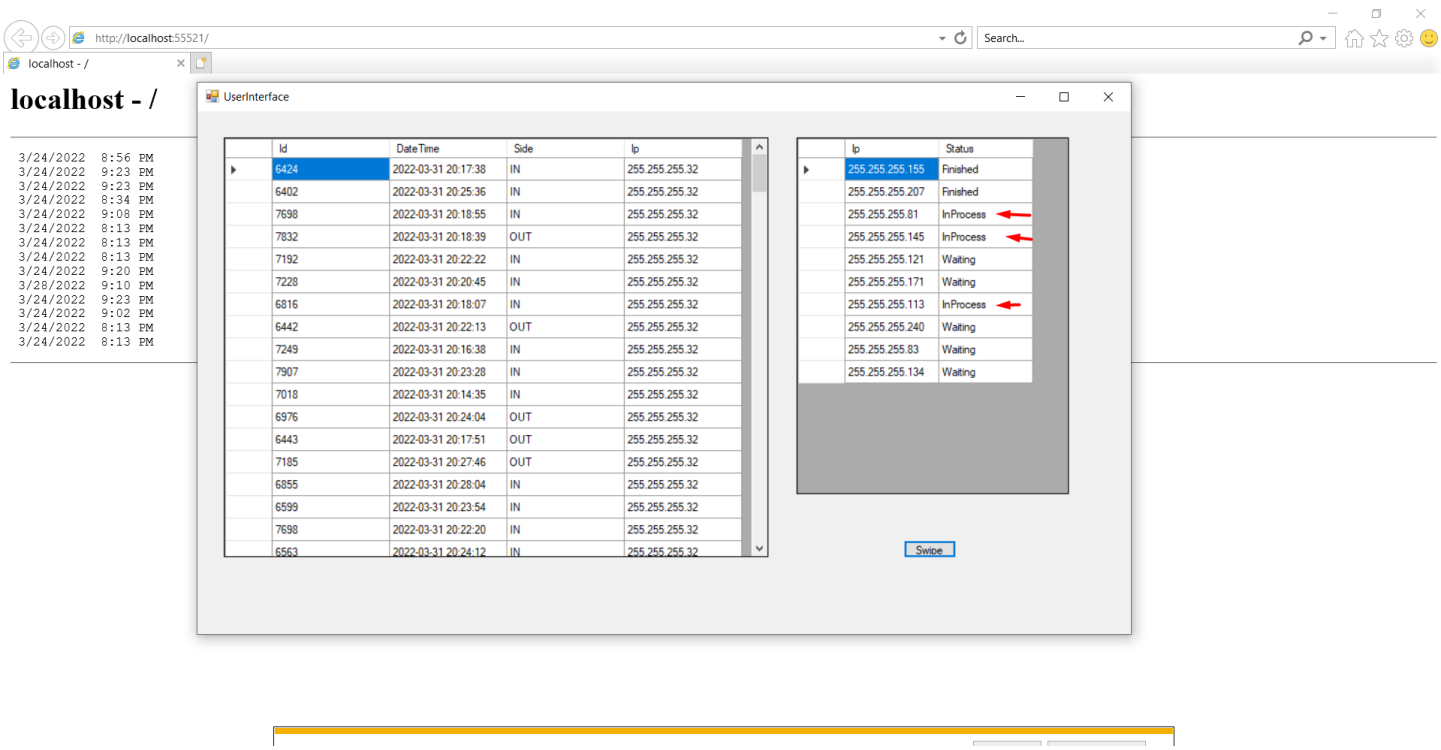


Current UML diagram shows the relationship between objects of the application with all fields and functions. There are 2 models called Swipe and Terminal. Using these models, application creates numerous terminals and Swipe records to save into the database. In order to communicate with the Database, the Repository design pattern has been used. SwipeRepo and TerminalRepo implement corresponding interfaces with such functions as GetAll records (Terminal and Swipe respectively) and Create functionality. Service1 which is WCF service in turn injects the repository class with the interface ISwipeRepo to decouple models using Dependency Injection. It also contains several functions. CollectSwipes method is responsible for saving each swipe into the database. ExtractRows method makes sure that all data in the records could be shown in an understandable way by dividing the fields into rows. And last GetSwipe takes the data from components and returns it as a swipe with its fields like Id, datetime and others.



Here is the definition of the only table in the database. It is called Swipes, has 4 nullable fields and all the swipes are saved inside the following table.

This is how it looks when we run the application and start the process of collecting swipes:



On the left it shows the data (saved swipes) from the database, on the right it shows the threads statuses that are trying to make a swipe: Waiting, in process and finished. Besides that, in the background there is WCF service running which makes it possible to collect swipes.

Effect of Synchronization primitives

There are several synchronization primitives that are used in multithreading applications. They are implemented by atomic operations and utilize corresponding memory barriers so the ones who utilize these primitives do not do it by themselves. Here are the most popular primitives: mutexes, shared or exclusive locks, semaphores, and condition variables.

Mutexes

Mutex can be used to prevent simultaneous access to the shared resource by several processes or threads. Ownership of the mutex should be waited by each thread before it can execute the code that needs access to the shared resource. For instance, in cases where multiple threads share access to the database, mutex object can be used by threads to give access to only one thread at a time to modify data in the database.

There are 2 versions of the mutex primitive: Sleep mutexes and spin mutexes. Spin mutex is a simple spin lock. When a thread tries to acquire a lock that is held by another thread, the second thread will spin while waiting for the lock to be released. A context switch cannot be done while holding a spin mutex to avoid deadlocking if a thread possessing a spin lock is not being executed on a CPU and all other CPUs are spinning on that lock due to its spinning nature. The scheduler lock, which must be retained throughout a context switch, is an exception. To meet this requirement while still protecting the scheduler data structures, the scheduler lock is passed from the thread being switched out to the thread being switched in as a special case. Because the bottom half of the code that schedules threaded interrupts and runs non-threaded interrupt handlers also utilizes spin mutexes, spin mutexes must deactivate interrupts while they are held to prevent the bottom half of the code from deadlocking against the top half code on the current CPU. While holding a spin lock, disabling interrupts has the unwanted side effect of increasing interrupt latency.

When a thread blocks on mutex, the second mutex performs a context switch. This type of mutex is called sleep mutex. It is not susceptible to the deadlocks with spin locks since a thread that contests on a sleep mutex blocks instead of spinning. Sleep mutex does not need to disable interrupts when they are held to avoid the type of deadlocks with spin locks.

Condition Variables

While waiting for a condition, the following primitive provide a logical abstraction to block a thread. It locks the appropriate mutex, tests the condition, if the condition is false blocks the condition variable. Condition variables themselves do not contain actual condition to test. The mutex is passed as interlock while waiting for on a condition to avoid lost wakeups.

Shared / Exclusive Locks

Shared or Exclusive locks, which are also called sx locks, are used to provide simple reader / writer locks. As its name implies, multiple threads are able to hold a shared lock at the same time, however only one thread can hold an exclusive lock. In addition, no threads can hold a shared lock if one thread holds an exclusive lock.

Semaphores

Semaphores are used to control access to the pool of resources. WaitOne method is called by threads to enter the semaphore. This method is inherited from the WaitHandle class, and by calling the Release method they release the semaphore. Every time when a thread enters the semaphore the count on a semaphor is decremented, and when a semaphore is released by a thread the count is incremented. Subsequent requests block until other threads releases the semaphore when the count is equal to zero. The count will be equal to the maximum value specified during the creation of semaphore when all threads have already released the semaphore.

Efficiency of Concurrent Programming techniques

Concurrent execution can improve performance in three ways: lower latency (that is, make a unit of work execute faster); hide latency (that is, allow the system to continue executing work during a long latency operation); and increase throughput (that is, to make the system able to perform more work). Concurrency for latency reduction is highly problem-specific, as it necessitates a parallel technique for the task at hand. This is simple for some issues, particularly those found in scientific computing: work may be partitioned a priori, and many compute elements can be assigned to the task. However, many of these tasks are so parallelizable that they don't require the tight coupling of a shared memory, and they may often be executed more cost-effectively on a grid of small machines rather than a smaller number of highly concurrent ones. Unit of work being long enough in its execution to amortize the substantial costs of coordinating several elements is required for using concurrency to reduce latency. One can imagine using concurrency to parallelize a sort of 40 million elements, but a sort of 40 elements is unlikely to take enough compute time to pay the overhead of parallelism. To summarize, the degree to which concurrency can be used to reduce latency is determined by the problem rather than those attempting to solve it, and many critical problems are just not susceptible to it.

References

Cantrill, B. and Bonwick, J. (2008). Real-world concurrency. *Communications of the ACM*, 51(11), pp.34–39.

www.usenix.org. (n.d.). *Synchronization Primitives*. [online] Available at: https://www.usenix.org/legacy/publications/library/proceedings/bsdcon02/full_papers/baldwin/baldwin_html/node5.html [Accessed 27 Mar. 2022].