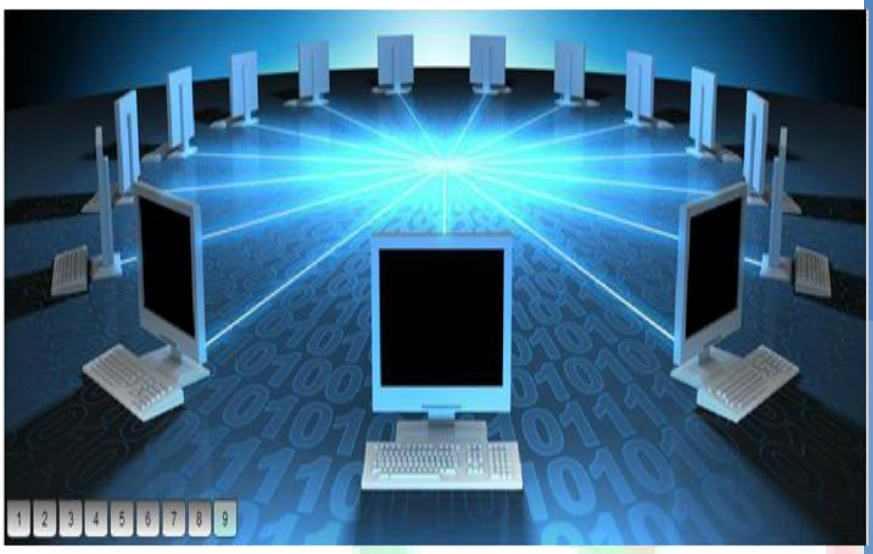
International Journal of Technology, Management & Humanities



www.ijtmh.com
ISSN (e) : 2454—566X

Volume - 1, Issue - 4
March - 2016

International Journal of Technology, Management and Humanities (IJTMH) refereed e-journal form in English.
International Journal of Technology, Management and Humanity is published on Quarterly basis with the aim to provide an appropriate platform presenting well considered, meaningful, constructively thought provoking and non-controversial but critically analyzing and synthesizing present and future aspects of Technical & scientific Education System with particular reference to the world.
The following types of article will be considered types of article will be considered

1. Research Articles: Original research in different fields of Science, Engineering and Management, Humanities will be evaluated as research articles.
2. Research Notes: These include articles such as manuscripts.
3. Reviews: Reviews of recent improvements, discoveries, developments, and thoughts in various fields of Science, management, and Engineering will be considered.
4. Frequency: FOUR issues in a year.

Indexing

JadounScience
http://www.jadounscience.com

International Society of Universal Research in Sciences

INDEX COPERNICUS INTERNATIONAL

DOAJ DIRECTORY OF OPEN ACCESS JOURNALS

Google Scholar

CABELL PUBLISHING, INC. Directories of Academic Journals

EBSCO HOST Electronic Journals Service

IJTMH

Contact Us
E-mail
submission@ijtmh.co
submissionijtmh@gmail.com
editor@ijtmh.com
editorijtmh@gmail.com

Condition Variable Implementation Using Semaphore in C#

Author

[1] Dr. Santosh Kumar Shukla, [2]Manjulata Shukla

[1](Associate Professor / Department of Computer Science/Savitribai Phule University, Pune/India)
[2] (Assistant Professor /Department of Computer Science/Savitribai Phule University, Pune /India)

Abstract : *The semaphore class works similar to the Monitor and Mutex class but lets you set a limit on how many threads have access to a critical section. It's often described as a nightclub (the semaphore) where the visitors (threads) stand in a queue outside the nightclub waiting for someone to leave in order to gain entrance. A critical section is a piece of code that accesses a shared resource (data structure or device) but the condition is that only one thread can enter in this section in a time.*

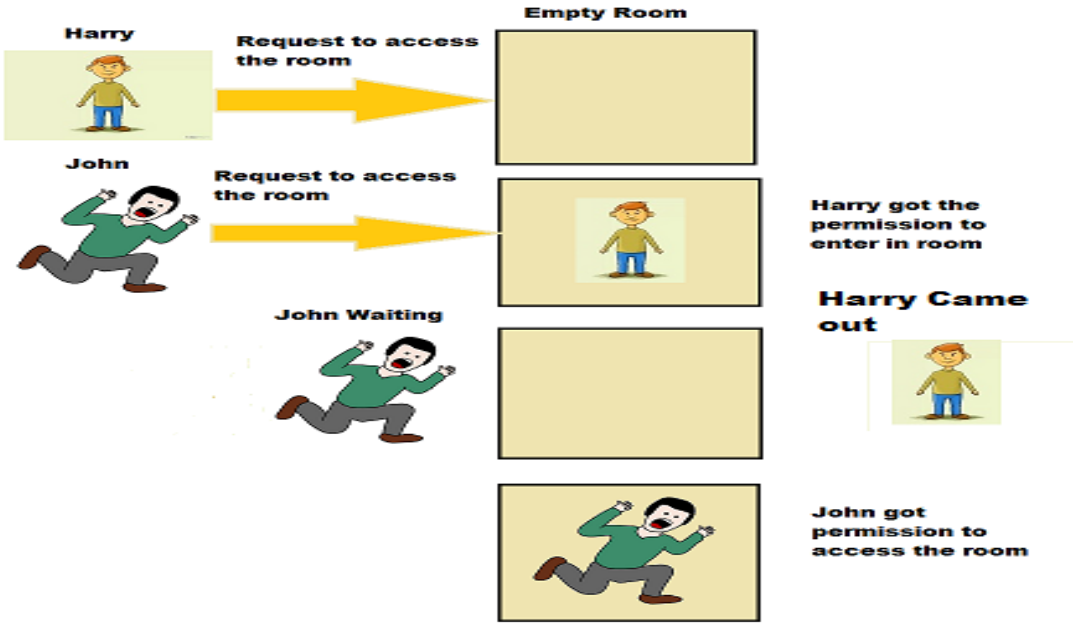
Keywords : *Critical Section, Reminder Section, Threading Semaphore, Mutex, Monitor*

1. Introduction

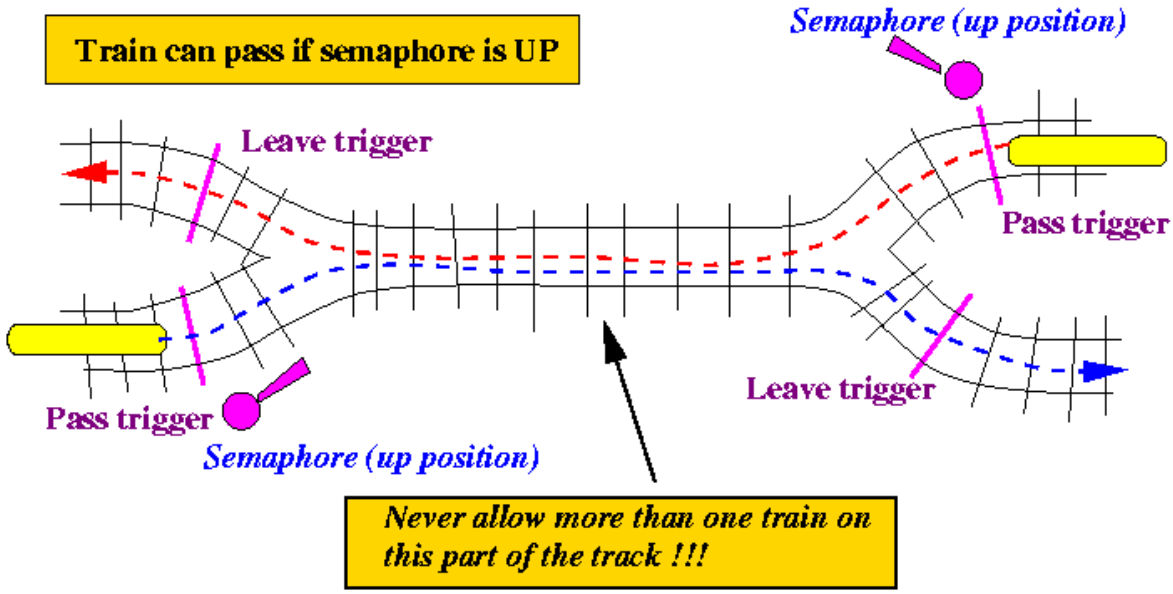
Semaphore provide all the method and property which are require to implement Semaphore. To use a semaphore in C#, you first need to instantiate an instance of a Semaphore object. The constructor, at a minimum, takes two parameters. The first is the number of resource slots initially available when the object is instantiated. The second parameter is the maximum number of slots available. If you want to reserve some slots for the calling thread, you can do so by making the first parameter smaller than the second. To reserve all slots for new threads, you should make both parameters the same.

After you instantiated your Semaphore object, you simply need to call the WaitOne() method when entering an area of code that you want restricted to a certain number of threads. When processing finishes, call the Release() method to release the slot back to the pool.
The count on a semaphore is decremented each time a thread enters the semaphore, and incremented when a thread releases the semaphore. When the count is zero, subsequent requests block until other threads release the semaphore. When all threads have released the semaphore, the count is at the maximum value specified when the semaphore was created.

For Example: Suppose we have one room in which only one person can enter at a time. watchman is sitting outside the room to maintained above condition. If one person comes to enter into the room he has to take permission from watchman first. If permission will be granted then only that person can enter into room.
if a person Harry is already there in room and some other guy John request watchman for getting permission to enter into room . Watchman will not grant access and ask the guy John to wait in wait queue. As soon as guy Harry will free the room watchman will give access to one of the guy waiting in wait queue. Here person Harry and John is like our process and watchman is like semaphore and room represents the resource .



Train Example



- The Binary Semaphore was invented by **E. W. Dijkstra** - a Dutch pioneer in Computer Science while he was directing the T.H.E. (Technical University of Eindhoven) Operating System project
- Dijkstra developed the semaphore concept when he studied the exclusive resource access problem:
    - Conflicting operations on a common resource (global variable) must happen serially - complete one operation before allowing another operation to even begin....
    - He notice that the real world already has a solution: Trains !
        - A train entering a "critical section" must pass a "semaphore" (signalling) device
        - If the semaphore is up, the train can pass, but as soon as the train crosses the semaphore, it trigger **all semaphore signals** to go down.

    - The semaphre signals will only be raised (to up) after the train leaves the critical section (triggers the "leave-trigger").
- The binary semaphore is an object that can take on 2 values: 0 (down) and 1 (up)

Some More Important Facts about Semaphore

- There is no guaranteed order, such as FIFO or LIFO, in which blocked threads enter the semaphore.
- The Semaphore class does not enforce thread identity on calls to WaitOne() or Release(). It is the programmer's responsibility to ensure that threads do not release the semaphore too many times.
- For example, suppose a semaphore has a maximum count of two, and that thread A and thread B both enter the semaphore. If a programming error in thread B causes it to call Release twice, both calls succeed. The count on the semaphore is full, and when thread A eventually calls Release, a SemaphoreFullException is thrown.
- Semaphores are of two types: local semaphores and named system semaphores.
- If you create a Semaphore object using a constructor that accepts a name, it is associated with an operating-system semaphore of that name.
- Named system semaphores are visible throughout the operating system, and can be used to synchronize the activities of processes.
- You can create multipleSemaphore objects that represent the same named system semaphore, and you can use the OpenExisting() method to open an existing named system semaphore.
- A local semaphore exists only within your process. It can be used by any thread in your process that has a reference to the local Semaphore object.
- Each Semaphore object is a separate local semaphore.

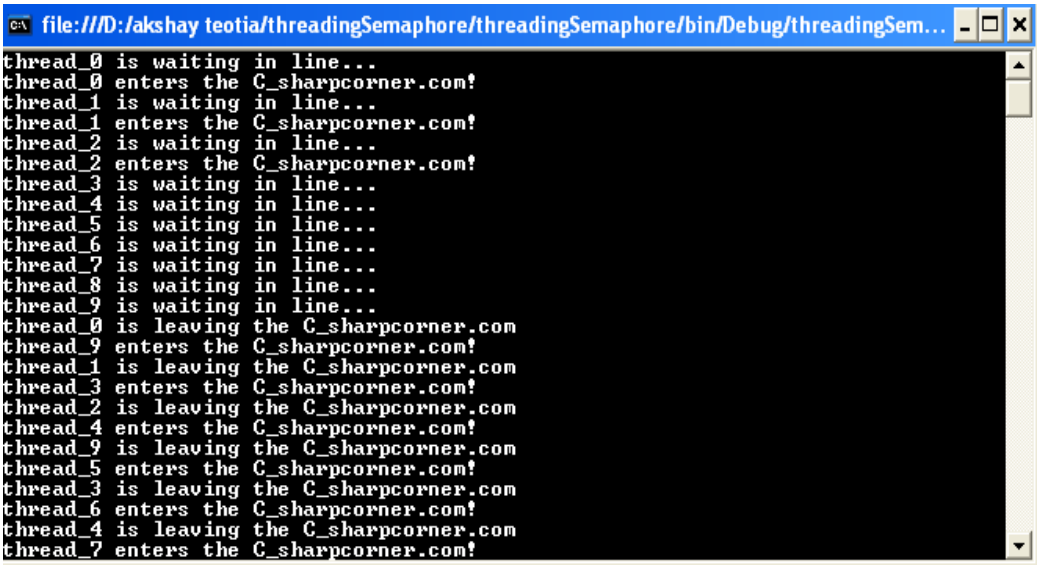
Creating a new semaphore is accomplished through one of the existing constructors:

- Semaphore(Int32,Int32)**:** Initializes a new instance of the Semaphore class,specifying the maximum number of concurrent entries and optionallyreserving some entries.
- Semaphore(Int32,Int32,String)**:** Initializes a new instance of the Semaphore class, specifying themaximum number of concurrent entries, optionally reserving some entriesfor the calling thread, and optionally specifying the name of a systemsemaphore object.
- Semaphore(Int32,Int32,String,Boolean)**:** Initializes a new instance of the Semaphore class, specifying themaximum number of concurrent entries, optionally reserving some entriesfor the calling thread, optionally specifying the name of a systemsemaphore object, and specifying a variable that receives a valueindicating whether a new system semaphore was created.
- Semaphore(Int32,Int32,String,Boolean,SemaphoreSecurity)**:** Initializes a new instance of the Semaphore class, specifying themaximum number of concurrent entries, optionally reserving some entriesfor the calling thread, optionally specifying the name of a systemsemaphore object, specifying a variable that receives a value indicatingwhether a new system semaphore was created, and specifying security accesscontrol for the system semaphore.

Using the code

```
using System;
using System.Threading;
namespace threadingSemaphore
{
class Akshay
   {
      static Thread[] threads = new Thread[10];
      static Semaphore sem = new Semaphore(3, 3);
      static void C_sharpcorner()
      {
         Console.WriteLine("{0} is waiting in line...", Thread.CurrentThread.Name);
         sem.WaitOne();
         Console.WriteLine("{0} enters the C_sharpcorner.com!", Thread.CurrentThread.Name);
         Thread.Sleep(300);
         Console.WriteLine("{0} is leaving the
C_sharpcorner.com", Thread.CurrentThread.Name);
         sem.Release();
      }
     static void Main(string[] args)
      {
         for (int i = 0; i < 10; i++)
         {
            threads[i] = new Thread(C_sharpcorner);
            threads[i].Name = "thread_" + i;
            threads[i].Start();
         }
        Console.Read();
       }
   }
}
```

Result



Conclusion

Well, history doesn't really have conclusions. But it does have a tendency to repeat. It will be nice if reading this anecdote prevents someone from repeating our mistakes, though I wouldn't bet on it. Implementing condition variables out of a simple primitive like semaphores is surprisingly tricky. The tricky part arises because of the binary atomic operation in Wait, where the lock is released and the thread is enqueued on the condition variable. If you don't have a suitable binary operation available, and you attempt to construct one by clever use of something like a semaphore, you'll probably end up with an incorrect implementation.

References

[1]. BIRRELL, A., GUTTAG, J., HORNING, J. AND LEVIN, R. Synchronization primitives for a multiprocessor: a formal specification. In Proceedings of the 11th Symposium on Operating System Principles (Nov. 1987), 94-102.
[2]. DIJKSTRA, E.W. The Structure of the T.H.E. Multiprogramming System. Commun. ACM 11, 5 (May 1968), 341-346
[3]. GOSLING, G., JOY, B., STEELE, G. AND BRACHA, G. The Java Language Specification, Second Edition, Sun Microsystems (2000), 429-447.
[4]. HOARE, C.A.R. Monitors: An operating system structuring concept. Commun. ACM 17, 10 (Oct.1974), 549-557.
[5]. SALTZER, J. Traffic control in a multiplexed computer system. Th., MAC-TR-30, MIT, Cambridge, Mass. (July 1966).