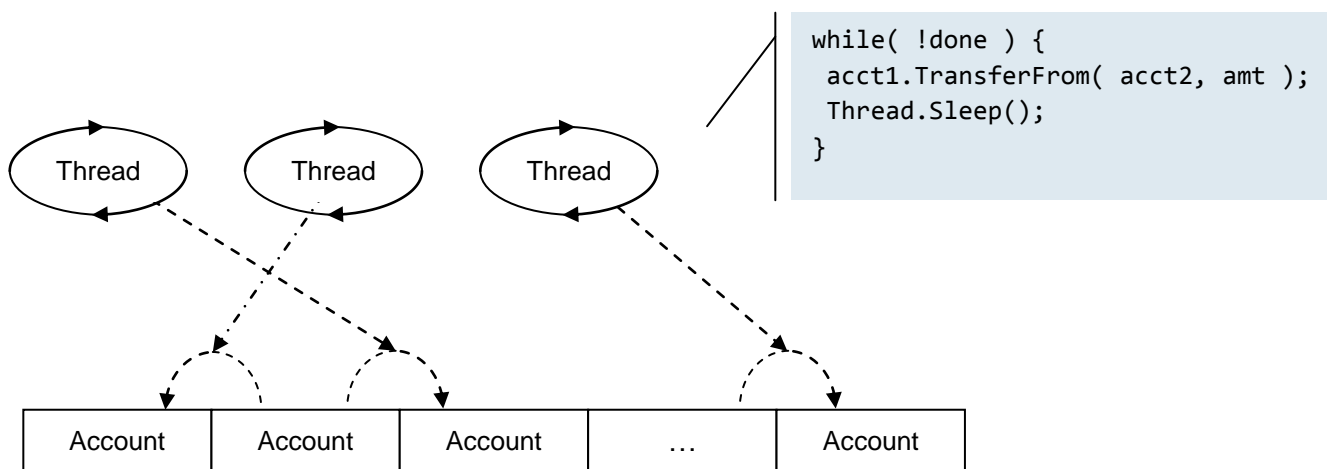


CS233S Lab 5

Part 1- Starting and Stopping Multiple Threads of Execution

In this activity you'll be working on a program that simulates a fairly real life multithreading scenario involving threads and thread synchronization. The simulation models a bank (consisting of an array of Account objects) that is servicing requests to transfer funds between accounts. In a real world scenario, these requests would be arriving via a web application or web service, which are carried out by different threads being managed by the runtime. In the simulation, you'll be creating these "transfer" threads yourself. When the simulation ends, you'll verify the internal consistency of each account. This means that when the simulation ends, the total amount of funds "on deposit" in the bank is the same as it was when the simulation started (since we're just transferring funds between accounts in the same bank). This design is illustrated by the following diagram:



Most of the code representing the logic of the simulation has already been provided for you. Your job will be broken down into two stages:

In the first stage, you'll write the code for the Main entry point that drives the simulation. This will involve creating and starting the transfer threads, letting them run for a prescribed amount of time, signaling these threads to stop the simulation, and the waiting for all of the threads to finish exiting. At this point, there will be a bug in the code as a result of having multiple threads update shared data structures without the benefit of any thread synchronization, but the bug won't be noticeable just yet.

In the second stage, you'll identify the problem areas resulting in thread synchronization issues that lead to corrupted accounts and use attempt to use the C# lock construct to make the code operate correctly.

For all parts of the lab you are expected to comment all the provided code as well as any you write yourself. Additionally, any time the lab asks a question you are to write your answer to it and submit those answers along with your final code.

steps:

1. Open the PartsOneAndTwo\app.sln solution provided to you by your instructor.

2. At the top of this program, there is a `SimulationParameters` class that encapsulates the operational parameters that drive the simulation. Review this class, and the comments associated with each constant, now.
3. Review the implementation of the `Account` class. Note that each account is assigned a unique number (0, 1, 2, ...) when it's instantiated, and that a public `TransferFrom` method is provided that allows callers to transfer funds from one account to another. Note that `TransferFrom` simply calls the private `Credit` and `Debit` methods to perform the actual transfer. Also note that the private `Credit` method makes a call to `Thread.Sleep`. This simulates in a consistent manner during the lab something that happens with far more irregularity in real world multithreaded programs. Do not remove this sleep, as it's intended to force a reproducible issue in the code that you'll be addressing later in the lab.
4. Locate the `Bank` class declaration. Review the following static variable declarations:
 - a. **bankAccounts**. This is the array of references to `Account` objects representing the bank.
 - b. **simulationOver**. This variable will be used by `Main` to signal to the other threads in the program that it's time to exit.
5. Locate the `Main` entry point of the program and note that there is a loop at the top of `Main` that initializes the `bankAccounts` array so that each element of the array refers to an instance of the `Account` class that's been instantiated with the initial deposit amount for each account.
6. After the bank account initialization loop, there should be two variables declared in `Main`: `transferThreads`, which is an array of references to the threads that will be performing the funds transfers; and `threadProc`, which is an instance of the `ThreadStart` delegate that's wired up to the `TransferThreadProc` method (you're going to use this same delegate multiple times in a loop you're about to write).

After these two variable declarations, write a loop that initializes each entry of the `transferThreads` array to refer to a new instance of the `Thread` class that has been initialized with the `threadProc` delegate. Inside your loop, assign each thread a unique name (something like "TX-0", "TX-1", etc). This will facilitate debugging later in the exercise. After assigning each thread a name, call the `Start` method on the new thread.

7. After the loop that creates and starts the transfer threads, call `Thread.Sleep` to block the main thread until `SimulationParameters.SIMULATION_LENGTH` milliseconds have elapsed.
8. After letting the simulation run for the specified amount of time, set the `simulationOver` variable to `true`. Having done this, wait for all of the simulation threads to exit looping through the `transferThreads` array, calling `Join` on each thread.
9. If you haven't already done so, locate and review the implementation of the `TransferThreadProc` method.
10. Build and run your program. Make sure your program can start and stop the threads correctly before proceeding. Insert code to print a message to the console in `Main` just before you set `simulationOver` to `true` indicating that you're about to stop the simulation. Then after using `Join` to wait for all of the simulation threads to exit, print another message to the console indicating that you've finished waiting for the other threads to exit.

Part 2- Experimenting with Thread Synchronization and Deadlock

Now that you've worked through the mechanics of starting and stopping threads, it's time to look at what kind of trouble you're in (and take some steps to prevent those troubles).

steps:

1. Locate and review the `VerifyAccounts` method that has been provided. This method loops through all of the accounts in the simulation, tallying the total funds that are on deposit. Once the funds have been tallied, a message is displayed to the console indicating the outcome of this "audit" operation. In order for our program to be operating correctly, this method should always indicate that the bank accounts are consistent.

Add a call to `VerifyAccounts` to the end of `Main` and rerun your program. You should find that some problems have been uncovered.

The problems happen because there are multiple threads performing updates on random accounts and, at times, more than one thread ends up trying to modify the same account that another thread is in the middle of modifying. Your task now is to use the locking primitives of CLR objects to prevent both of these problems.

2. Update `Account.TransferFrom` to acquire the `SyncBlock` associated with its own account instance, as well as the lock associated with the other account instance. You'll need to use two separate, nested `lock` constructs to do this. Only after acquiring both locks is it safe to perform the credit and debit operations on the two accounts.

Note: if you're feeling a little uncomfortable about grabbing two different locks here, that's good – but keep going. If you're not feeling any discomfort at the prospect of making two separate lock acquisitions here, you will momentarily.

3. Build and rerun your program. Does `VerifyAccounts` now indicate that the accounts are consistent when the simulation ends? Does the simulation end at all? If you run the program repeatedly, does the simulation always end? If so, does that mean it always will run to completion?
4. Your `TransferFrom` method should currently look something like this:

```
lock( this ) {
    lock( otherAcct ) {
        // Calls to Credit and Debit here...
    }
}
```

Making two calls to acquire locks like this is extremely dangerous since any thread executing this code might get preempted after acquiring the first lock, but before acquiring the second lock. If another thread were to then attempt to transfer funds between the same two accounts, but in the opposite direction, deadlock could ensue.

To make this dangerous situation more obvious, insert a call to `Thread.Sleep(10)` between the two calls to `lock`, as follows:

```
lock( this ) {
    Thread.Sleep(10);
    lock( otherAcct ) {
        // Calls to Credit and Debit here...
    }
}
```

Rebuild and rerun your program (multiple times if necessary). It shouldn't take you too long to deadlock your program such that the simulation never ends.

If you aren't already doing so, run the program under the debugger (F5). When the transfer threads deadlock and fail to respond to the main thread's attempt to shut them down, use the Debug/Break menu item to break into the program using the debugger. Open the watch window and add two watches on `this.AccountNumber` and `otherAcct.AccountNumber`. Now use the threads display on the location control bar to cycle between the threads of the application (Main, TX-0, TX-1, TX-2, and TX-3). Can you identify which threads are deadlocked, versus the threads that are simply stuck waiting to acquire their first lock? From run to run of your program, you can expect to find a 2-, 3-, or even 4-way deadlock among the transfer threads. Note as you cycle between threads the debugger will show you the current instruction pointer, but that it will be positioned on the line after the spot where the thread is stuck. So if the debugger indicates that the instruction pointer is currently positioned on the call to `Thread.Sleep(10)`, that thread is really stuck acquiring its first lock.

5. While the C# `lock` construct is fairly easy to use when you're only acquiring one lock primitive at a time, it's dangerous to use whenever you find yourself acquiring multiple locks like we were doing above, as was just demonstrated.

What we need is a way to eliminate making multiple discrete calls to acquire a lock, and instead replace them with a single call to the runtime that requests ownership of multiple locks simultaneously. The runtime will then acquire all of the specified locks on our behalf without putting our thread in the position of holding one lock while waiting for ownership of another. This kind of functionality is not supported by the built-in `SyncBlock` architecture, but is supported using a different locking primitive called a **Mutex** (where "Mutex" is a contraction of "mutual exclusion").

At a very high level, the `System.Threading.Mutex` class represents the exact same kind of primitive as the built-in `SyncBlock`; namely, a primitive that represents a lock which may only be acquired by one thread at a time. However, the `Mutex` class has the added distinction of being derived from the `System.Threading.WaitHandle` class. The `WaitHandle` class represents an abstraction for blocking one or more threads until an arbitrary condition has been achieved (referred to as 'signaled'). Another example of a `WaitHandle`-derived type is used in asynchronous delegate invocation – namely the `IAAsyncResult.AsyncWaitHandle` property (which returns an instance of an "event" object that is signaled when the method call completes). In the case of a `Mutex`, it becomes signaled when no threads have ownership of the mutex, and unsignaled when one thread does (thereby blocking any other thread that attempts to acquire the same mutex).

What makes the `Mutex` class so useful in terms of deadlock prevention is that references to more than one instance of the `Mutex` class can be grouped together in an array and passed to the static

WaitHandle.WaitAll method. This capability provides the support we need for making one method call to the runtime with a request to perform a deadlock-free acquisition of more than one lock.

6. Add a private Mutex member variable to the Account class and initialize the mutex inline where you declare it:

```
private Mutex acctLock = new Mutex();
```

7. If you are using the lock construct in the Balance property (you should be), switch to calling WaitOne on the mutex instead, as shown here:

```
public double Balance
{
    get
    {
        double b = 0;

        if( acctLock.WaitOne() )
        {
            b = balance;
            acctLock.ReleaseMutex();
        }

        return(b);
    }
}
```

8. Update TransferFrom so that it declares and initializes a local variable array of Mutex references so that it contains a reference to the account's mutex as well as the mutex for the account funds are being transferred from (otherAcct), as shown below:

```
Mutex[] acctLocks = { this.acctLock, otherAcct.acctLock };
```

9. Replace the two lock constructs in `TransferFrom` with a single call to `WaitHandle.WaitAll`, passing in the array of mutex references initialized in the previous step. You'll also need to release ownership of each mutex in the array as a separate step (assuming `waitAll` succeeds). Here's what your code should look like:

```
if( WaitHandle.WaitAll(acctLocks) )
{
    try
    {
        otherAcct.Debit(amt);
        this.Credit(amt);
    }
    finally
    {
        foreach( Mutex m in acctLocks )
        {
            m.ReleaseMutex();
        }
    }
}
```

10. You should no longer be using the `lock` construct anywhere in the application.
11. Rebuild the application and then run it multiple times. The program should always produce correct results without deadlocking.

Part 3- Delegate-based Asynchronous Invocation

In this part of the activity you'll revise the program to use delegates and the built-in thread pool to achieve asynchronous execution, instead of creating and controlling threads yourself.

Note that in this part of the exercise, you'll start by opening an existing project containing the starting point for the application.

steps:

1. Open the `PartThree\app.sln` solution provided to you by your instructor.
2. Note that the simulation parameters enumeration has been modified slightly to represent the fact that you'll no longer be creating your own threads.
3. Note that the `Account` class is identical to the solution from the previous exercise using mutexes with one exception. Because we'll be using pooled threads in this activity, the diagnostic calls to `Console.WriteLine` now show the thread ID with the output as opposed to the thread name that was used in the previous part of the exercise. Although not as readable as names, it will still be useful later in this exercise for indicating how many unique threads from the pool were involved in a run of the program.
4. Locate the `Bank` class. Note that after the `bankAccounts` member variable, a new member variable named `PendingTransferCount` has been declared and initialized to 15. This version of the program

will perform a fixed number of transfers, as opposed to allowing the simulation to run for a preset amount of time.

5. Since you'll be using delegates to invoke the `TransferFrom` method on the `Account` class, you'll need to define a custom delegate type that matches its signature. After the declaration of the `PendingTransferCount` variable, and before the `Main` entry point, define a delegate named `TransferProc` suitable to call `Account.TransferFrom`.
6. Locate the loop in `Main` that creates the accounts used in the simulation. This loop is followed by another loop that will choose a random transfer amount, choose two separate random accounts to operate on, and then issue a request to the thread pool (using `BeginInvoke`) to call the `TransferFrom` method on the account that will be credited funds. All of the requests will be queued up to the thread pool immediately, and then handled by the thread pool asynchronously as it sees fit.
7. Locate the position inside the loop where a `TODO` comment indicates you need to initialize a delegate that refers to the `TransferFrom` method. Do so now using the `TransferProc` delegate you defined earlier. Make sure you initialize the delegate to refer to the `TransferFrom` method of the account that's going to have funds credited to it (`creditAccount`).
8. Use the delegate you just initialized to issue a request to the thread pool by calling `BeginInvoke`. Because the simulation needs to know when the last transfer occurs, you're going to need to request that the thread pool threads call you back after each transfer completes.

The method that you want to arrange to have called each time a transfer completes has already been written - `TransferComplete`. Locate and review its implementation now. Note that it simply decrements `PendingTransferCount` (using the thread-safe `Interlocked.Decrement` method). Go back to `Main` and note that after the loop that issues the requests to the thread pool, `Main` simply spins until `PendingTransferCount` reaches zero; indicating that the simulation is complete.

Back in the loop that calls `BeginInvoke`, initialize a new instance of the standard `AsyncCallback` delegate so that it refers to the `TransferComplete` method. Pass this delegate to the `BeginInvoke` method. You may also initialize the `AsyncCallback` delegate outside (and before) the loop, since you'll be referring to the same `TransferComplete` method in every call to `BeginInvoke`; but that's just a minor performance improvement.

9. Build and run the program. Barring errors in your use of delegates, the program should run to completion and still show consistent accounts. But how many unique threads from the thread pool were actually transferring funds between accounts?
10. Update `TransferFrom` so that it calls `Thread.Sleep(750)`; between the calls to `Credit` and `Debit` on the two accounts. Build and run the program again. Now how many unique threads are involved with transferring funds between the accounts? What does this tell you about the heuristics that the thread pool manager is using to trigger the creation of new threads in the pool?