Industrial Programming

Lecture 5: C# Threading Introduction, Accessing Shared Resources

Based on: “An Introduction to programming with C# Threads”
By Andrew Birrell, Microsoft, 2005

Examples from “Programming C# 5.0”, Jesse Liberty, O'Reilly. Chapter 20.

Processes and Threads

- Traditionally, a process in an operating system consists of an execution environment and a single thread of execution (single activity).
- However, *concurrency* can be required in many programs (e.g in GUIs) for various reasons.
- The solution was to improve the notion of a process to contain an execution environment and *one or more threads* of execution.

Processes and Threads (cont'd)

- An *execution environment* is a collection of kernel resources locally managed, which threads have access to. It consists of:
  - An address space.
  - Threads synchronization and communication resources
  - Higher-level resources such as file access.

Processes and Threads (cont'd)

- Threads represent activities which can be created and destroyed dynamically as required and several of them can be running on a single execution environment.
- The aim of using multiple threads in a single environment is:
  - To maximise the concurrency of execution between operations, enabling the *overlap of computation* with input and output.
  - E.g. one thread can execute a client request while another thread serving another request (optimising server performance).

Concurrency and Parallelism

- In some applications concurrency is a natural way of structuring your program:
  - In GUIs separate threads handle separate events
- Concurrency is also useful operating slow devices including e.g. disks and printers.
  - IO operations are implemented as a separate thread while the program is progressing through other threads.
- Concurrency is required to exploit multi-processor machines.
  - Allowing processes to use the available processors rather than one.

Sources of Concurrency (cont'd)

- Concurrency aides user interaction:
  - Program could be processing a user request in the background and at the same time responding to user interactions by updating GUI.
- Concurrency aides performance:
  - A web server is multi-threaded to be able to handle multiple user requests concurrently.

Threads Primitives

- Thread Creation.
- Mutual Exclusion.
- Event waiting.
- Waking up a thread.
- The above primitives are supported by C#’s *System.Threading* namespace and C# lock statement.

Thread Creation

- A thread is constructed in C# by:
  - Creating a *Thread* object.
  - Passing to it a *ThreadStart* delegate.
  - Calling the *start* method of the created thread.
- Creating and starting a thread is called *forking.*

Thread Creation Example

```
Thread t = new Thread(new
  ThreadStart(func.A));
t.start();
func.B();
t.join();
```

- The code above executes functions *func.A()* and *func.B()* concurrently.
- Thread *t* is created and started.
- Execution completes when both method calls have completed.

Mutual Exclusion

- Mutual exclusion is required to control threads access to a shared resource.
- We need to be able to specify a region of code that only one thread can execute at any time.
- Sometimes called critical section.

C# Mutual Exclusion Support

- Mutual exclusion is supported in C# by class *Monitor* and the *lock* statement.

  lock(expression)

  statement
- The *lock* argument can be any C# object.
- By default, C# objects are *unlocked.*
- The *lock* statement
  - locks the object passed as its argument,
  - executes the statements,
  - then unlocks the object.
- If another thread attempts to access the locked object, the second thread is blocked until the lock releases the object.

Example: Swap

```
public void Swap() {
 lock (this) {
   Console.WriteLine("Swap enter: x={0}, y={1}",
                     this.x, this.y);
   int z = this.x;
   this.x = this.y;
   this.y = z;
   Console.WriteLine("Swap leave: x={0}, y={1}",
                     this.x, this.y);
 }
}
```

Examples from “Programming C# 3.0”, Jesse Liberty, O'Reilly. Chapter 20.

Example: Swap

```
public void DoTest() {
   Thread t1 = new Thread(
                     new ThreadStart(Swap));
   Thread t2 = new Thread(
                     new ThreadStart(Swap));

   t1.Start();
   t2.Start();
   t1.Join();
   t2.Join();
 }
```

Waiting for a Condition

- Locking an object is a simple scheduling policy.
  - The shared memory accessed inside the *lock* statement is the scheduled resource.
- More complicated scheduling is sometimes required.
  - Blocking a thread until a condition is true.
  - Supported in C# using the *Wait, Pulse* and *PulseAll* functions of class *Monitor.*

Waiting for a Condition (cont'd)

- A thread must hold the lock to be able to call the *Wait* function.
- The *Wait* call unlocks the object and blocks the thread.
- The *Pulse* function awakens at least one thread blocked on the locked object.
- The *PulseAll* awakens all threads currently waiting on the locked object.
- When a thread is awoken after calling *Wait* and blocking, it re-locks the object and return.

Example: Increment/Decrement

```
public void Decrementer() {
  try {
    // synchronise this area
    Monitor.Enter(this);
    if (counter < 1) {
      Console.WriteLine("[{0}] In Decrementer. Counter: {1}.
Waiting...",
               Thread.CurrentThread.Name, counter);
      Monitor.Wait(this);
    }

    while (counter > 0) {
      long temp = counter;
      temp--;
      Thread.Sleep(1);
      counter = temp;
      Console.WriteLine("[{0}] In Decrementer. Counter:{1}.",
               Thread.CurrentThread.Name, counter);
    }    } finally {
      Monitor.Exit(this);
    }  }
```

Example: Increment/Decrement

```
public void Incrementer() {
 try {
   // synchronise this area
   Monitor.Enter(this);

   while (counter < 10) {
     long temp = counter;
     temp++;
     Thread.Sleep(1);
     counter = temp;
     Console.WriteLine("[{0}] In Incrementer.{1}.",
             Thread.CurrentThread.Name, counter);
    }
   Monitor.Pulse(this);
 } finally {
   Console.WriteLine("[{0}] Exiting ...",
           Thread.CurrentThread.Name);
   Monitor.Exit(this);
 }   }
```

Example: Increment/Decrement

```
public void DoTest() {
     Thread[] myThreads = {
   new Thread( new ThreadStart(Decrementer)),
   new Thread( new ThreadStart(Incrementer)) };

   int n = 1;
   foreach (Thread myThread in myThreads) {
     myThread.IsBackground = true;
     myThread.Name = "Thread"+n.ToString();
     Console.WriteLine("Starting thread {0}", myThread.Name);
     myThread.Start();
     n++;
     Thread.Sleep(500);
   }
   foreach (Thread myThread in myThreads) {
     myThread.Join();
   }
   Console.WriteLine("All my threads are done");
   }
private long counter = 0;
```

Example Explained

- 2 threads are created: one for incrementing another for decrementing a global counter
- A monitor is used to ensure that reading and writing of the counter is done atomically
- Monitor.Enter/Exit are used for entering/leaving an atomic block (critical section).
- The decrementer first checks whether the value can be decremented.
- Monitor.Pulse is used to inform the waiting thread of a status change.

Thread Interruption

- Interrupting a thread is sometimes required to get the thread out from a wait.
- This can be achieved in C# by using the *interrupt* function of the *Thread* class.
- A thread *t* in a wait state can be interrupted by another thread by calling *t.interrupt().*
  - *t* will then resume execution by relocking the object (maybe after waiting for the lock to become unlocked).

Interrupts complicates programs and should be avoided if possible.

Race Conditions

Example:

Thread A opens a file

Thread B writes to the file

→ The program is successful, if A is fast enough to open the file, before B starts writing.

Deadlocks

Example:

Thread A locks object M1

Thread B locks object M2

Thread A blocks trying to lock M2

Thread B blocks trying to lock M1

→ None of the 2 threads can make progress

Avoiding Deadlocks Involving Locks

- Maintain a partial order for acquiring locks in the program.
  - For any pair of objects {M1, M2}, each thread that needs to have both objects locked simultaneously should lock the objects in the same order.
  - E.g. M1 is always locked before M2.

→ This avoids deadlocks caused by locks.

Deadlock Caused By Wait

- Example:

Thread A acquires resource 1

Thread B acquires resource 2

Thread A wants 2, so it calls Wait to wait for 2

Thread B wants 1, so it calls Wait to wait for 1

- Again, partial order can be used to avoid the deadlock.

Other Potential Problems

- Starvation: When locking objects or using *Monitor.Wait()* on an object, there is a risk that the object will never make progress.
- Program complexity.