Threading

Cpt S 321

Washington State University

What most of you are used to

- Code executing line-by-line in sequence
- Functions completing entirely before returning

```
F()
{
   Statement_A
   F2(); // F2 completes execution entirely before moving on to statement B
   Statement_B
}
```

Threads

- A “thread” (in the context of coding) is an execution context (or path) that executes independently
- Analogy:
  - Processes run in parallel within the OS and are controlled by the OS
  - Threads run in parallel with a process and are managed by that process
- Threads in a C++, C# or Java application have shared access to (almost) all the memory in the process
- Each thread has its own call stack.

Why Threads are Good

- Can’t do certain things in a user-friendly way without them
  - Imagine if your web browser locked up and didn’t process any user input while transferring data over the web. Could never stop loading of a particular page or close the browser while the page was downloading.
  - Any computational task that takes over a second or two would prevent the user interface from refreshing and make the user think the app is frozen. It takes CPU time just to produce the interface that you see in a WinForms or other GUI application.

Why Threads are Good

- Faster execution
  - If the computing device has 4 processors on it, then you can process 4 things at once in parallel
  - Can also take 1 task and split it up into 4 parts, making the task go 4X faster than if you execute it “in one piece”

Why Threads are Good

- In some cases simpler code
  - Imagine if you had to write code that rapidly flipped back and forth between updating your GUI and downloading data from a network. This would result in coupling the logic of network IO and GUI if you are not careful.
  - With threads you just have one thread for the GUI and another for network IO and each has execution logic only relating to its specific task

Why Threads are Difficult

- Executing code in parallel introduces an entirely new class of problems
- Single-threaded application:

```
if (some_shared_variable == true)
{
    can assume some_shared_variable stays true while executing here
}
```

- Mutli-threaded application:

```
if (some_shared_variable == true)
{
    some_shared_variable could change at any point during the execution of this method
}
```

How to Create a Thread

- In C++ and C# the concept is identical:
  1. Create a thread object (std::thread in C++, Thread class in C#)
  2. Give the thread a reference to a method
  3. Start the thread

- The thread will start executing the method you specified and at the same time the code that created the thread will continue to run

Thread Methods

- Start
  - Starts the thread
  - Not many guarantees in terms of timing here, other than “some time after” this function is called the thread starts running
- Join
  - If thread t1 calls t2.Join(), then thread t1 blocks until thread t2 completes
  - It’s the calling thread that blocks
  - A blocking call is a function call that makes the caller wait (or “**block**”) until the action is complete before continuing
- Sleep (static method)
  - Suspends the thread for the specified amount of time. The thread is **blocked**

Simple example

```
using System;
using System.Threading;
class ThreadTest
{
   static void Main()
   {
      Thread t = new Thread (WriteY);     // Kick off a new thread
      t.Start();                          // running WriteY()

      // Simultaneously, do something on the main thread.
      for (int i = 0; i < 1000; i++) Console.Write ("x");
   }
   static void WriteY() {
      for (int i = 0; i < 1000; i++) Console.Write ("y");
   }
}
```

```
// Typical Output:
xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
…
```

Useful properties

- IsAlive: returns true, until the point where the thread ends. A thread ends when the delegate passed to the Thread's constructor finishes executing.
- Name: You can set a name for each thread to make debugging easier. You can set a thread's name just once; attempts to change it later will throw an exception.
- CurrentThread: Static property that allows you to get the thread that us currently executing.
- ThreadState: Tells us the execution state of a thread. Use for diagnostic, not for synchronization.

Another example - Join

```
static void Main()
{
    Thread t = new Thread(Go);
    t.Start();
    t.Join();
    Console.WriteLine("Thread t has ended!");
}
static void Go() { for (int i=0;i<3;i++) Console.WriteLine(”Y"); }
```

```
// Output:
Y
Y
Y
Thread t has ended!
```

Another example - Thread.Sleep

- Thread.Sleep (TimeSpan.FromHours(1)); // Sleep for 1 hour
- Thread.Sleep (500); // Sleep for 500 milliseconds

Local State

```
// Output:
?
?
?
?
```

```
static void Main()
{
   new Thread(Go).Start();     // Call Go() on a new thread
   Go();                       // Call Go() on the main thread
}

static void Go() {
   // Declare and use a local variable - 'cycles'
   for (int cycles = 0; cycles < 2; cycles++) Console.WriteLine ('?');
   // A separate copy of cycles is created on each thread's memory stack
}
```

Shared State

```
// Likely output:
Done
```

```
class ThreadTest {
   bool _done;  // a field
   static void Main() {
      ThreadTest tt = new ThreadTest();     // Create a common instance
      new Thread(tt.Go).Start();
      tt.Go();
   }
   void Go() // Note that this is an instance method
   {
      if (!_done) { _done = true; Console.WriteLine ("Done"); } }
   }
}
```

// Because both threads call Go() on the same ThreadTest instance, they share the **_done** field. This results in “Done” likely being printed once instead of twice.

Shared State (cont.)

```
// Likely output:
Done
```

```
class ThreadTest {
   static void Main() {
      bool done = false;
      ThreadStart action = () =>
      {
         if (!done) { done = true; Console.WriteLine ("Done"); }
      };
      new Thread (action).Start();
      action();
   }
}
```

// **Local variables captured by a lambda expressions** are converted by the compiler into **fields**, and so can also be shared

Shared State (cont.)

```
// Likely output:
Done
```

```
class ThreadTest {
   static bool _done;
   static void Main()
   {
      new Thread (Go).Start();
      Go();
   }
   static void Go() { if (!_done) { _done = true; Console.WriteLine ("Done"); } }
}
// Static fields are shared between all threads
```

The examples showing shared state…

- The output is actually indeterminate
- Let’s try running those examples. What is your output?
  1. Done?
  2. Done
     Done?
- Now swap the two statements in the Go method and try again. What is your output?
  1. Done?
  2. Done
     Done?
- Swapping the statements increases the likelihood that Done is printed twice: one thread can be evaluating the if statement while the other thread is executing the WriteLine statement before it has a chance to set _done to true.

C# lock Keyword

- http://msdn.microsoft.com/en-us/library/c5kehkcz.aspx

- Done on an object -> Creates a scope -> Ensures that only one thread will be executing in that scope for that object at any given time

- Used to solve the issue mentioned on the previous slide

- Syntax is simple:

lock (object) { … }

C# **lock** Keyword

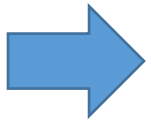
- Assume an application has two threads: A and B

```
void F(object o)
{
  lock (o) {
     // If thread A is executing in here and thread B reaches the
     // lock statement above, then thread B will either:
     // 1. Halt and wait for thread A to complete the lock scope if it's locking on the same object
     // 2. Enter the scope concurrently with thread A if it's locking on a different object
  }  // End of lock scope
}
```

Locking and Thread Safety

```
// Output:
Done
```

Thread-safe code



```
class ThreadSafe {
   static bool _done;
   static readonly object _locker = new object();
   static void Main() {
      new Thread (Go).Start();
      Go(); }
   static void Go()
   {
      lock (_locker) { // must be a reference type object
         if (!_done) {
            Console.WriteLine ("Done");
            _done = true;
         }
      }
   }
}
```

Problems with locking

- Can forget to lock
- Can lead to deadlocking: two threads may end up waiting for each other indefinitely