Threading and Unsafe Code

Mingsheng Hong
CS 215, Spring 2008

Threading Overview

- Threading provides concurrent execution
  - As opposed to "sequential" execution
  - Useful on uniprocessors and multiprocessor
- How to create a thread
  - `System.Threading` namespace
  - New `Thread` instance with `ThreadStart` delegate
    - `public delegate void ThreadStart();`
  - `Start` method to start the thread

Thread Operations

- `Thread.Abort` method
  - Terminate a thread
  - Raise `ThreadAbortException`
  - Can be supressed with `Thread.ResetAbort()`
- `Thread.Sleep` method
- `Thread.Join` method
  - Wait for the completion of another thread
  - Better than busy-polling on `Thread.IsAlive`
- `Thread.Priority` property

Thread Synchronization

- Pre-emptive thread scheduling
- When two threads access the same data
- For example

```
public int Increment(ref int x) { return ++x; }
```

  - What happens when called by two threads?

Thread Synchronization

- Synchronization primitives
  - Way to ensure that only one thread executes code in a region at once
  - Called "critical section"
- C# provides (mostly in System.Threading)
  - lock statement
  - Monitor class
  - Interrupt
  - several others (see Birrell's paper or MSDN)

Lock

- Basic idea: each object has a lock

```
public int Increment(ref int x) {lock(this) return ++x;}
```

  - lock prevents more than one thread from entering
  - forces sequential order
- What should we lock on?
  - for instance variables: `this`
  - for globals and statics: `typeof(container)`
  - something that will be same for all threads that access this shared memory

ThreadPool

- Instead of explicitly creating threads
  - create a pool
  - enqueue jobs with `QueueUserWorkItem`
    - takes a `WaitCallback` delegate, to be passed to worker threads
- Good for large amounts of parallel work
  - "embarrasingly parallel" problems
  - automatically scales to number of processors

ThreadPool Example

```
namespace ThreadPoolExample {
class Program {
   static void Main(string[] args) {
        for (int i = 0; i < 20; i++) {
              ThreadPool.QueueUserWorkItem(
                    new WaitCallback(DoWork), i);
        }
   }

   static void DoWork(object state) {
        int threadNumber = (int)state;
        Console.WriteLine("Thread {0} reporting.", state);
   }
}
}
```

Pointer Types

- A separate category of types
- A pointer is a variable whose value is a memory address
- Common operations on pointers
  - `int x = 10; int* px = &x; *px = 5;`
  - `AStruct* pa = …; pa->mf; (*pa).mf`
  - `pa++, pa--, pa + pb, pa > pb`
  - Conversion
    - to `void*`, or from `null`
    - to and from integral types

Unsafe Mode

- Sometimes need access to pointers
  - e.g. access to OS, memory mapped device, or implement time-critical algorithms
- use `unsafe` modifier
  - ```
    unsafe static void swap(int* x, int* y) {
      int tmp = *x;
      *x = *y;
      *y = tmp;
    }
    ```
  - ```
    int x = 0, y = 1;
    unsafe { swap(&x, &y); }
    ```

Pointer Parameters

- Can pass pointers as `ref` or `out` parameters

```
unsafe public void Method(out int* pi) {
     int i = 10;
     fixed(int* pj = &i) {
          pi = pj;
     }
}
```

- what is wrong with this method?

Pointer Details

- Can only refer to value types
  - cannot refer to a reference
  - cannot refer to a struct that contains references
- No pointer arithmetic allowed on `void*`
- `stackalloc` gets memory from the stack
- Note `int* pi, pj;  // NOT int *pi, *pj;`

Compiler Setting

- Turn on the /unsafe compiler flag
- Or open Visual Studio project properties