Locks

Recall

- Recall our example from last time: The output was not 3000000

```
using System.Threading;
using System.IO;
using System;
class MainClass{
    static int result = 0;
    static void worker(){
        for(int i = 0; i < 1000000; ++i)
            result++;
    }
    public static void Main(string[] args){
        Thread t1 = new Thread( () => { worker(); } );
        Thread t2 = new Thread( () => { worker(); } );
        Thread t3 = new Thread( () => { worker(); } );
        t1.Start(); t2.Start(); t3.Start();
        t1.Join(); t2.Join(); t3.Join();
        Console.WriteLine("Foo: " + result);
    }
}
```

Why?

- Recall: CPU doesn't necessarily do increment as atomic operation
  - Load
  - Modify
  - Store

Race Condition

- Race condition (or correctness hazard): Correctness of program depends on scheduling order of two or more threads
  - If thread A “wins the race”: One set of results
  - If thread B wins: Different results
- All programs with race conditions are erroneous

Hazards

- When do they occur?
  - When contention for shared resource
  - Locals are never shared
  - So: If program only uses locals: No contention
- Note: Static instance variables are inherently shared
- Note: If you pass same reference to two threads: Shared resource

Globals

- Most useful programs must use global resources
  - Shared data (ex: Large array – too costly to duplicate)
  - Common I/O device (terminal)
  - Common disk file
  - We often use globals to communicate between threads
- So it's hard to avoid potential for race conditions

Visibility

- Another concern: Visibility
- Suppose we have code like so:

```
static bool flag1=false;
static bool flag2=false;
```

```
void func1(){
    doSomething();
    flag1=true;
    while(flag2 == false )
        ;
    doSomethingElse()
}
```

```
void func2(){
    while(flag1 == false )
        ;
    doStuff();
    flag2=true;
    finishUp();
}
```

Problems

- First problem: Busy waiting
  - Wastes CPU
  - Creates heat
  - Burns battery life
- Second problem: No guarantee that this code does what we want
  - As long as code is correct from single thread's viewpoint, instructions can be reordered
  - So: Legal for compiler (or hardware) to reorder the write of flag1 until after the while loop
- If we replaced assignments with:

  Interlocked.Increment(flag1)

  or

  Interlocked.Increment(flag2)
- Then: Code works. Why?

Reasoning

- Concept: Interlocked operations impose *sequential consistency*
- What's that?

Definition

- Define a relation: *happens-before*
- If statement X happens-before statement Y, it means... What you'd think!
  - Results of X are visible when Y begins
  - Written $X \to Y$
- If we don't know that $X \to Y$ and we don't know that $Y \to X$ then we say that X and Y are *concurrent*
  - Written $X \parallel Y$

Consider

- Consider previous code
- Which statement(s) happen before which other statements?

Result

- Here's all we can say:
  - $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$
  - $f \rightarrow g \rightarrow h \rightarrow i \rightarrow j$
- Hmm. Notice we can say nothing with regard to inter-thread operations
- The two threads are entirely *concurrent*
  - $a \parallel f$, $a \parallel g$, $a \parallel h$, $b \parallel f$, $b \parallel g$, etc.
- This is different when interlocked operations are used...

Interlocked

- If we use interlocked operations, this creates a synchronization point
- Suppose threads A and B access *the same atomic variable x*
  - Thread A reads x at statement α
  - Thread B writes x at statement β
    - Then exactly one of these will be true:
      $\alpha \rightarrow \beta$
      $\beta \rightarrow \alpha$
  - This is the crucial inter-thread tie that we need!

New Code

```
static int flag1=0;
static int flag2=0;
```

```
void func1(){
    doSomething();
    //(a)
    Interlocked.Increment(flag1); //(b)
    while( Interlocked.Add(flag2,0) == 0 )
        ;
    //(c)
    doSomethingElse()
}
```

```
void func2(){
    while(Interlocked.Add(flag1,0) == 0 ) //(e)
        ;
    doStuff();
    //(f)
    Interlocked.Increment(flag2); //(g)
    finishUp();
    //(h)
}
```

Relationships

- $a \rightarrow b \rightarrow c \rightarrow d$
- $e \rightarrow f \rightarrow g \rightarrow h$
- We know that $b \rightarrow f$
  - Why? Because:
    - $e \rightarrow f$
    - We get past e iff flag1 is nonzero
    - b is the only place we set flag1 to nonzero
    - Thus, $(b||e) \rightarrow f$
- Likewise, we know $f \rightarrow d$
  - Why? Because $f \rightarrow g \rightarrow d$
  - Do you see why?

Interlocked

- Interlocked solves some visibility and ordering problems
- But: Important drawbacks:
  - Don’t help with busy waiting
  - Difficult to reason about program correctness
  - Don’t help with ensuring several interdependent variables are kept consistent

Mutex

- Mutex = MUTual EXclusion
  - Also called a lock
- A mutex is essentially a boolean with two operations: lock and unlock

Mutex

- Lock: Pseudocode:

```
while( locked == true )
    releaseCpu();
locked=true;
```

- What's not shown here is that all operations are atomic
  - Locked can't be changed after leaving the loop but before setting locked=true
  - We normally can't write code like that ourselves; needs OS/runtime support

Mutex

- Unlock: Pseudocode:

```
locked=false;
```

Vocabulary

- If thread A successfully calls mutex lock (i.e., when lock() returns), we say A has *acquired* the mutex (or ”it has locked the lock”)
- If A calls unlock(), we say it has *released* the mutex (or ”unlocked the lock”)

Vocabulary

- What happens if A acquires the lock and then B calls lock()?
  - OS takes B off the CPU until A releases the mutex
  - We say B is *blocked* on the mutex (or B is ”waiting for the lock”)
  - The CPU is free for other threads while B is blocked
  - So we *are not* busy waiting

Syntax

- In C#, to perform mutex operation: We first create an object and ensure all threads can see it:
  object myLock = new object()
- Then, we write:

```
lock( myLock ){
    ...
}
```

  - Lock will be held for duration of brace-block

Sequential Consistency

- Mutexes (mutices?) provide some sequential consistency guarantees as well

```
static object myLock = new object();
```

```
void func1(){
    //(a)
    lock(myLock){
        //(b)
     doSomething();
        //(c)
    }
    //(d)
}
```

```
void func2(){
    //(e)
    lock(myLock){
        //(f)
     doSomething();
        //(g)
    }
    //(h)
}
```

Ordering

- Trivially, we know:
  $a \rightarrow b \rightarrow c \rightarrow d$
  $e \rightarrow f \rightarrow g \rightarrow h$
- Since we are locking on same lock, we know either:
  $d \rightarrow f$ or
  $h \rightarrow b$
- So we can reason about inter-thread dependencies this way

Rule of Thumb

- To make life easier, we have some common patterns we use with mutexes
- Pattern: If you have a shared variable: you guard it with a mutex
  - Note: Every thread must use the same mutex to get any useful synchronization
  - In practice, that means your mutex object will almost always be static

Incorrect

```
void func1(){
    object M = new object();
    lock(M){
        ...
    }
}
```

- M is totally useless!

Example

- Suppose we have a network server
- Remote machines contact it, request resources, get data back
- Suppose it takes time to compute response, so we cache last request to save time

Example

- Example non-threaded server implementation:

```
class Server{
    static DataItem lastItem;
    static string lastIdentifier;
    DataItem handleRequest(string identifier){
        if( identifier == lastIdentifier )
            return lastItem;
        else{
            lastItem = loadDataFromDisk(identifier);
            lastIdentifier = identifier;
            return lastItem;
        }
    }
}
```

- Clearly this would not be multi-thread-safe. Why not?

Solution

```
class Server{
    static DataItem lastItem;
    static string lastIdentifier;
    static object M = new object();
    DataItem handleRequest(string identifier){
        lock(M){
            if( identifier == lastIdentifier )
                return lastItem;
            else{
                lastItem = loadDataFromDisk(identifier);
                lastIdentifier = identifier;
                return lastItem;
            }
        }
    }
}
```

Extension

- Now, we decide to extend it: Save the last five requests in the cache. The non-threaded code:

```
class Server{
    static DataItem[] lastItem = new DataItem[5];
    static string[] lastIdentifier = new string[5];
    DataItem handleRequest(string identifier){
        for(int i=0;i<5;++i){
            if( identifier[i]== lastIdentifier[i] )
                return lastItem[i];
        }
        //FIFO replacement strategy
        for(int i=0;i<4;++i){
            lastItem[i] = lastItem[i+1];
            lastIdentifier[i] = lastIdentifier[i+1];
        }
        lastItem[4] = loadDataFromDisk(identifier);
        lastIdentifier[4] = identifier;
        return lastItem[4];
    }
}
```

Threaded

- Sprinkle some magic locking pixie dust around...Broken!

```
class Server{
    static DataItem[] lastItem = new DataItem[5];
    static string[] lastIdentifier = new string[5];
    object M = new object();
    DataItem handleRequest(string identifier){
        lock(M){
            for(int i=0;i<5;++i){
                if( identifier[i]== lastIdentifier[i] ) return lastItem[i];
            }
            //FIFO replacement strategy
            for(int i=0;i<4;++i){
                lastItem[i] = lastItem[i+1];
                lastIdentifier[i] = lastIdentifier[i+1];
            }
            lastItem[4] = loadDataFromDisk(identifier);
            lastIdentifier[4] = identifier;
            return lastItem[4];
        }
    }
}
```

Problem

- If one thread is computing, it blocks all other threads from doing any work
  - Even if they could go immediately!
- How to fix?

Solution?

- How about this?

```
class Server{
    static DataItem[] lastItem = new DataItem[5];
    static string[] lastIdentifier = new string[5];
    object M = new object();
    DataItem handleRequest(string identifier){
        for(int i=0;i<5;++i){
            if( identifier[i]== lastIdentifier[i] ) return lastItem[i];
        }
        lock(M){
            //FIFO replacement strategy
            for(int i=0;i<4;++i){
                lastItem[i] = lastItem[i+1];
                lastIdentifier[i] = lastIdentifier[i+1];
            }
            lastItem[4] = loadDataFromDisk(identifier);
            lastIdentifier[4] = identifier;
            return lastItem[4];
        }
    }
}
```

Nope

- We're accessing shared data without holding the lock!

Fixed? (Nope!)

- Is this good? Why not?

```
class Server{
    static DataItem[] lastItem = new DataItem[5];
    static string[] lastIdentifier = new string[5];
    object M = new object();
    DataItem handleRequest(string identifier){
        lock(M){
            for(int i=0;i<5;++i)
                if( identifier[i] == lastIdentifier[i] ) return lastItem[i];
            for(int i=0;i<4;++i){
                lastItem[i] = lastItem[i+1];
                lastIdentifier[i] = lastIdentifier[i+1];
            }
        }
        var tmp = loadDataFromDisk(identifier);
        lock(M){
            lastItem[4] = tmp;
            lastIdentifier[4] = identifier;
        }
        return lastItem[4];
```

Problem

- Race condition after releasing lock but before returning lastIdentifier[4]

Solution (Finally!)

- Finally!

```
class Server{
    static DataItem[] lastItem = new DataItem[5];
    static string[] lastIdentifier = new string[5];
    object M = new object();
    DataItem handleRequest(string identifier){
        lock(M){
            for(int i=0;i<5;++i){
                if( identifier[i]== lastIdentifier[i] )
                    return lastItem[i];
            }
        }
        var tmp = loadDataFromDisk(identifier);
        lock(M){
            //FIFO replacement strategy
            for(int i=0;i<4;++i){
                lastItem[i] = lastItem[i+1];
                lastIdentifier[i] = lastIdentifier[i+1];
            }
            lastItem[4] = tmp;
            lastIdentifier[4] = identifier;
            return lastItem[4];
        }
    }
}
```

Deadlocks

- Deadlocks are one of the main hazards when working with mutexes
- Suppose we have two global variables, each protected by its own lock

```
static object xLock = new object();
static object yLock = new object();
static int x,y;
```

```
void func1(){
    lock(xLock){
        if( x > 5 ){
            lock(yLock){
                y++;
            }
        }
    }
}
```

```
void func2(){
    lock(yLock){
        if( y > 100 ){
            lock(xLock){
                x++;
            }
        }
    }
}
```

Problem

- classic x-y / y-x locking pattern
- This puts you on the express train to Deadlockville

Solutions

- One solution: Number all locks
- Then: Ensure locks are acquired in increasing numerical order
  - Ex: Let xLock be 0 and yLock be 1
  - If you want both locks, you must grab xLock first
- Problem: Second thread doesn't know right away if it needs xLock
  - Maybe y is 42
  - Lower resource utilization
    - Need to grab lots of locks "just in case" we need them later
  - Hard to remember the number of each lock

Assignment

- Extend the previous lab:
  - Every half second, print the status of each downloaded item: Either “Complete” or “In progress” or “Error”
- When all items are downloaded (or have errored out), exit.
- Turn in your CS files only
- More details follow...

Assignment

- Your code must have no race conditions, no possibility of incorrect output, and no visibility hazards
- Your program must not crash; make sure to handle all exceptions
- Ex: Your display might look like this:

```
http://www.example.com: In progress
http://www.example.org/foo/bar: Complete
http://www.example.net/abc.txt: In progress
http://www.example.org/def.gif: Error
----------------------------------------------
http://www.example.com: In progress
http://www.example.org/foo/bar: Complete
http://www.example.net/abc.txt: Complete
http://www.example.org/def.gif: Error
----------------------------------------------
http://www.example.com: Complete
http://www.example.org/foo/bar: Complete
http://www.example.net/abc.txt: Complete
http://www.example.org/def.gif: Error
```

Sources

- B. Goetz et al. Java Concurrency in Practice. Addison-Wesley.
- M. Herlihy &; N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers.
- David Beazley & Brian K. Jones. Python Cookbook. O'Reilly Media.
- Varun. C++ 11 Multithreading Part 5: Using Mutex to Fix Race Conditions. http://thispointer.com/c11-multithreading-part-5-using-mutex-to-fix-race-conditions/
- http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf

Created using $\LaTeX$.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org