Async and Parallel Programming: Application Design

Understanding the Dangers of Concurrency

Joe Hummel, PhD

joe-hummel@pluralsight.com

http://blog.joehummel.net/

@joehummel

pluralsight

hardcore developer training

Overview

- **Your presenter: Joe Hummel, PhD**
  - PhD in field of high-performance computing
  - An exciting time to be working in this area…

- **Agenda for this module:**
  - *The many pitfalls when parallel programming*
  - *The most common pitfall — race conditions*
  - *Locking vs. Lock-free*
  - *Synchronization primitives*
  - *Concurrent Data Structures*
  - *Lots of demos*

Beware the pitfalls of concurrency

- **Async and Parallel programming are full of potential pitfalls…**
  - *Race conditions*
  - *Starvation*
  - *Livelock*
  - *Deadlock*
  - *Optimizing compilers*
  - *Optimizing hardware*

  - For a quick read on the subject:
    - "*Tools And Techniques to Identify Concurrency Issues*", R. Patil and B. George, MSDN Magazine, June 2008

    Online: http://msdn.microsoft.com/en-us/magazine/cc546569.aspx

Correctness?

- **Your code should provide 2 guarantees:**

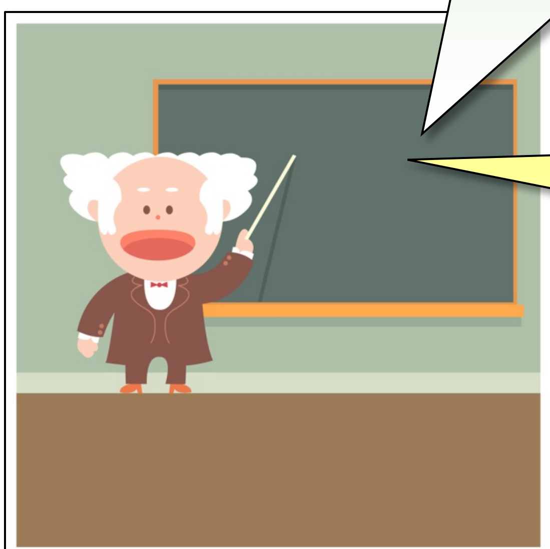
Safety: *nothing bad happens.*

Liveness: *eventually something good happens.*

Important Terminology

Def: a race condition exists when the outcome depends on the timing of events.

Race conditions are bad !



Def: a critical section is smallest region of code involved in a race condition.

Identify and resolve…

The danger of shared resources

- Most common pitfall — concurrent access to shared resources
  - variables
  - objects
  - collections
  - files
  - …

Danger == conc
Safe == concurr

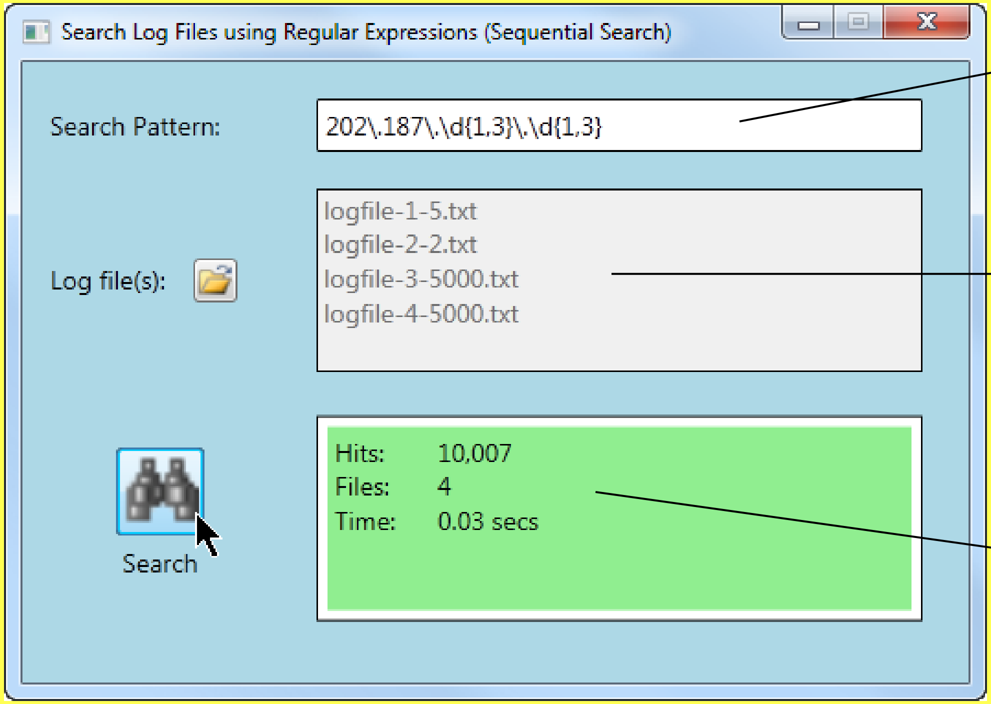
```
int sum = 0;    // Parallel code:

Task.Factory.StartNew(() =>
  { sum = sum + obj1.Computation(); });

Task.Factory.StartNew(() =>
  { sum = sum + obj2.Computation(); });

Task.Factory.StartNew(() =>
  { sum = sum + obj3.Computation(); });
```

Error!!

Danger == concurrent reads & writes, or concurrent writes;
Safe == concurrent reads.

DEMO

- **Searching log files…**



search string as a regular expression (e.g. IP address `202.187.*.*`*)*

files to search…

results

Solutions?

- Redesign to eliminate shared resources / critical sections
  - e.g., each task uses only local memory (aka "*task local storage*")

- Use thread-safe entities within critical sections
  - e.g., TPL offers thread-safe data structures (*System.Collections.Concurrent*)

ConcurrentBag *ConcurrentDictionary* *BlockingCollection*
ConcurrentStack *ConcurrentQueue*

- Use synchronization to control access within critical sections

CountdownEvent / ManualResetEvent / AutoResetEvent
Lock
Barrier *Monitor* *Semaphore*
Interlocked
SpinLock / SpinWait *Mutex*

Solution #1: Lock

- **Surround critical section with a lock on a common object**
  - *restricts entry to one task at a time*

Sequential

```
int sum = 0;

sum = sum + obj1.Computation();
sum = sum + obj2.Computation();
```

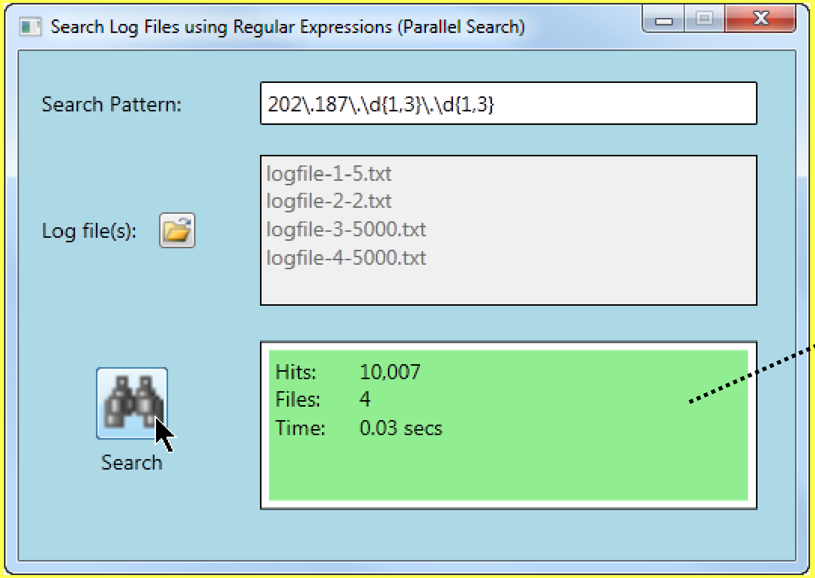
Parallel

```
int sum = 0;
var l   = new object();   // common object:

Task.Factory.StartNew(() =>
  {
     int t = obj1.Computation();
     lock(l) { sum += t; }
  });

Task.Factory.StartNew(() =>
  {
     int t = obj2.Computation();
     lock(l) { sum += t; }
  });
```

Locking is a general technique based on .NET Monitor class

DEMO

- Searching log files in parallel using a lock…




```
lock (l)
{
  hits++;
}
```

Solution #2: Interlocked

- A HW-based lock for simple, arithmetic critical sections
  - *typically more efficient*

Sequential

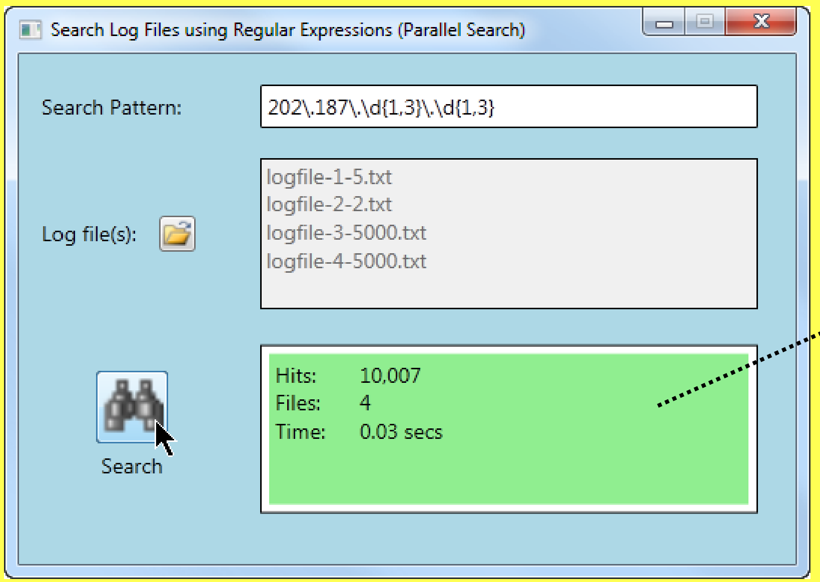
```
int sum = 0;

sum = sum + obj1.Computation();
sum = sum + obj2.Computation();
```

Parallel

```
using System.Threading;

int sum = 0;

Task.Factory.StartNew(() =>
  {
      int t = obj1.Computation();
      Interlocked.Add(ref sum, t);
  });

Task.Factory.StartNew(() =>
  {
      int t = obj2.Computation();
      Interlocked.Add(ref sum, t);
  });
```

DEMO

- Searching log files in parallel using interlocking…




```
Interlocked.Increment(ref hits);
```

Solution #3: Lock-free

- Locking leads to contention, hindering performance
- Use lock-free[1] designs whenever possible…
- Example:
  - *use **local resource** instead of shared resource*
  - ***return** result instead of updating shared resource*
  - *join, **combine** partial results into final result*
  - *no locking in this case*

Local memory ("task local state")

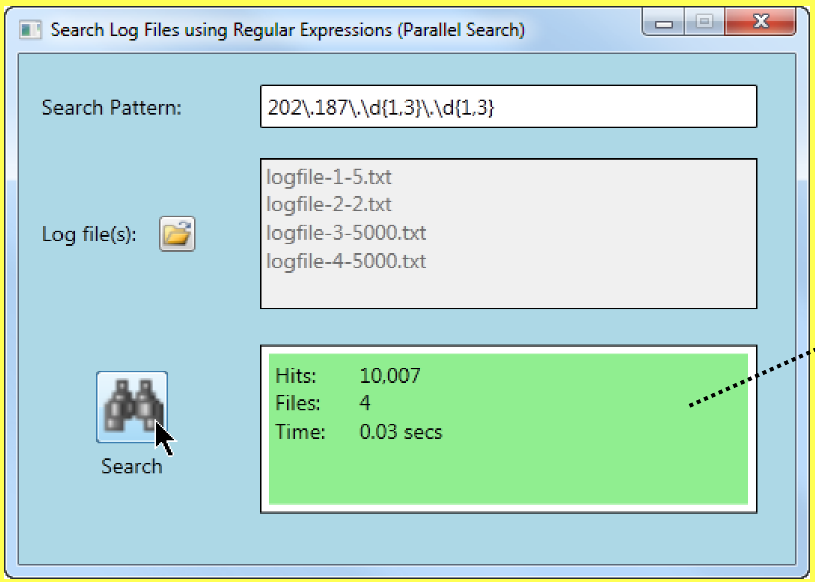
```
var t1 = Task.Factory.StartNew<int>(() =>
  {
      int t = obj1.Computation();
      return t;
  });

var t2 = Task.Factory.StartNew<int>(() =>
  {
      int t = obj2.Computation();
      return t;
  });

sum = t1.Result + t2.Result;
```

Wait… Harvest partial results, produce final result

[1] Lock-free is not necessarily freedom from locks, but freedom from arbitrary waiting…

DEMO

- Searching log files in parallel using a lock-free approach…




```
int lHits = 0;
  .
  .
  .
lHits++;
  .
  .
  .
return lHits;
```

Another danger…

- **Another form of race condition:**
  - *shared objects and **non-thread-safe** classes…*

Random and List are *not* thread-safe…

Sequential

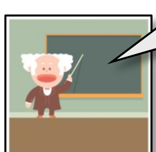
```
List<int> results = new List<int>();
Random    rand    = new Random();

for (int i=0; i < N; i++)
{
  int r = RunSimulation( rand );
  results.Add(r);
}
```

Parallel

```
List<int> results = new List<int>();
Random    rand    = new Random();

for (int i=0; i < N; i++)
{
  Task.Factory.StartNew(() =>
    {
      int r = RunSimulation( rand );
      results.Add(r);
    }
  );
}
```

Error!!

__Def__: an object or method is thread-safe if parallel use does not cause a race condition.



Solution?

- *Locking is one solution…*
- *Other solutions?*

Sequential

```
List<int> results = new List<int>();
Random    rand    = new Random();

for (int i=0; i < N; i++)
{
  int r = RunSimulation(rand);
  results.Add(r);
}
```

thread-safe
data structure

```
using System.Collections.Concurrent;

var results = new ConcurrentQueue<int>();

for (int i=0; i < N; i++)
{
  Task.Factory.StartNew(() =>
  {
   // use random seed so tasks generate different sequences:
   var rng = new RNGCryptoServiceProvider();
   byte[] data = new byte[4];
   rng.GetBytes(data);
   int seed = BitConverter.ToInt32(data, 0);
   var rand = new Random(seed);

   var rand = new Random();

   int r = RunSimulation(rand);
   results.Enqueue(r);
  });
}
```

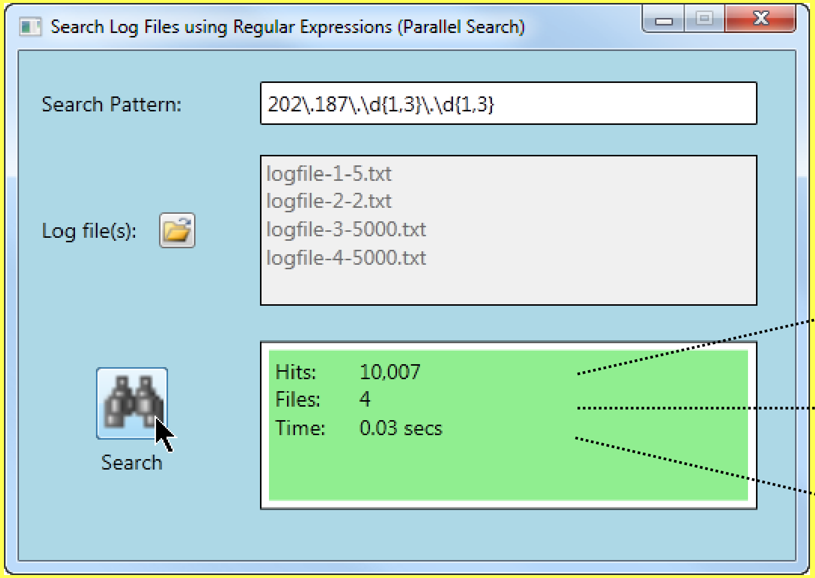
yields
duplicates

local instance

Correct solutions can be subtle, and hard to test!

DEMO

- **Thread-safety** when searching log files in **parallel**…



single, shared Regex object? √

File.ReadAllBytes? √

Encoding.UTF8.GetString? ?

Synchronization primitives in brief…

- **TPL builds upon existing .NET synchronization primitives:**

Primitive	Purpose
Monitor	*General-purpose .NET synchronization class*
Lock	*Enforces one-at-a-time semantics using Monitor class*
Mutex	*Win32 lock suitable for inter-process sync ("mutual exclusion")*
Interlocked	*Hardware-based lock for simple, arithmetic operations*
Semaphore	*Enforces N-at-a-time semantics via Win32*
SpinLock / SpinWait	*Lock-like mechanisms that loop ("spin") instead of yield CPU*
Barrier	*Allows tasks to synchronize ("sync-up") before start of next phase*
CountdownEvent, ManualResetEvent, AutoResetEvent	*Additional ways for tasks to synchronize with one another…*

Goal of TPL is to provide better abstractions so you *don't* need all these…

Concurrent Data Structures

- **TPL offers a set of thread-safe data structures:**
  - `ConcurrentBag<T>`
  - `ConcurrentQueue<T>`
  - `ConcurrentStack<T>`
  - `ConcurrentDictionary<T>`
  - `BlockingCollection<T>`

Redesigned and optimized for concurrent access --- we'll discuss in future lecture ("Designs and Patterns")

A few words about performance…

- Future lecture on *Designs and Patterns* will discuss in more detail
- But there's a few lessons now worth mentioning…

I/O is *hard* to parallelize

- *Requires support from OS and HW*
- *Don't expect much from commodity HW*

Experiment…

- *Try different ideas*

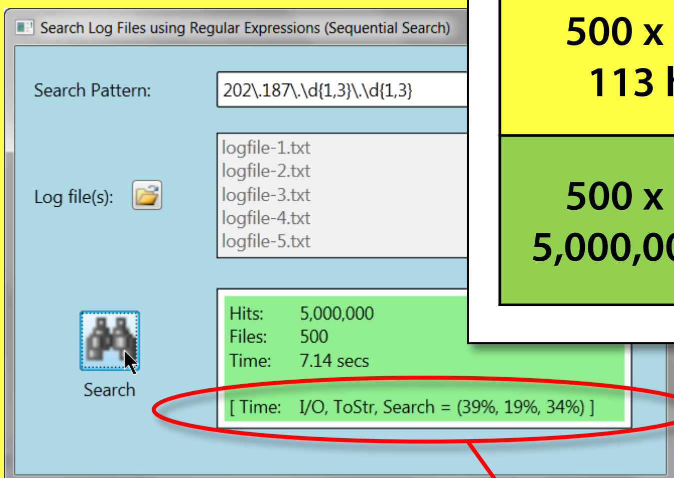
Instrument your app

- *Time things*
- *Measure things*

Amdahl's law

- *Performance limited by sequential component*

DEMO: performance of log file search app



Input	Sequential	Parallel-LockFree	
		version 1	*version 2*
500 x 1MB 113 hits			
500 x 1MB 5,000,000 hits			

shared RE object

local RE object

Instrumentation: % of I/O, ToString, Search

Summary

- Beware the many dangers of async and parallel programming:

RACE CONDITIONS
LIVELOCK
DEADLOCK
OPTIMIZING COMPILERS
STARVATION
OPTIMIZING HARDWARE

- The most common pitfall: race conditions from shared resources

- Various solutions…
  - *redesign to eliminate*
  - *use thread-safe entities*
  - *use synchronization*

most preferred

least preferred

References

- **Microsoft's main site for all things parallel:**
  - http://msdn.microsoft.com/concurrency
- **MSDN technical documentation:**
  - http://tinyurl.com/pp-on-msdn
- **Background reading on the dangers of concurrency, and solutions:**
  - Just about any textbook on *Operating Systems*
  - *Concurrent Programming on Windows*, by Joe Duffy, Addison-Wesley
  - *Win32 Multithreaded Programming*, by A. Cohen and M. Woodring