# Data Races and Locks

Unit 2.a

# Acknowledgments

- Authored by
  - Sebastian Burckhardt, MSR Redmond

# Concepts

Correctness Concept

- Atomicity Violations
- Data Races
- Data-Race-Free Discipline
- Immutability
- Isolation
- Deadlocks

Code Concept

- Lock(myobj)

Part 1

# ATOMICITY VIOLATIONS

CS15

# Atomic, Informally

A statement sequence S is *atomic* if
S's effects appear to other threads as if
S executed without interruption

**Atomicity Violation: An error caused by unexpected lack of atomicity.**

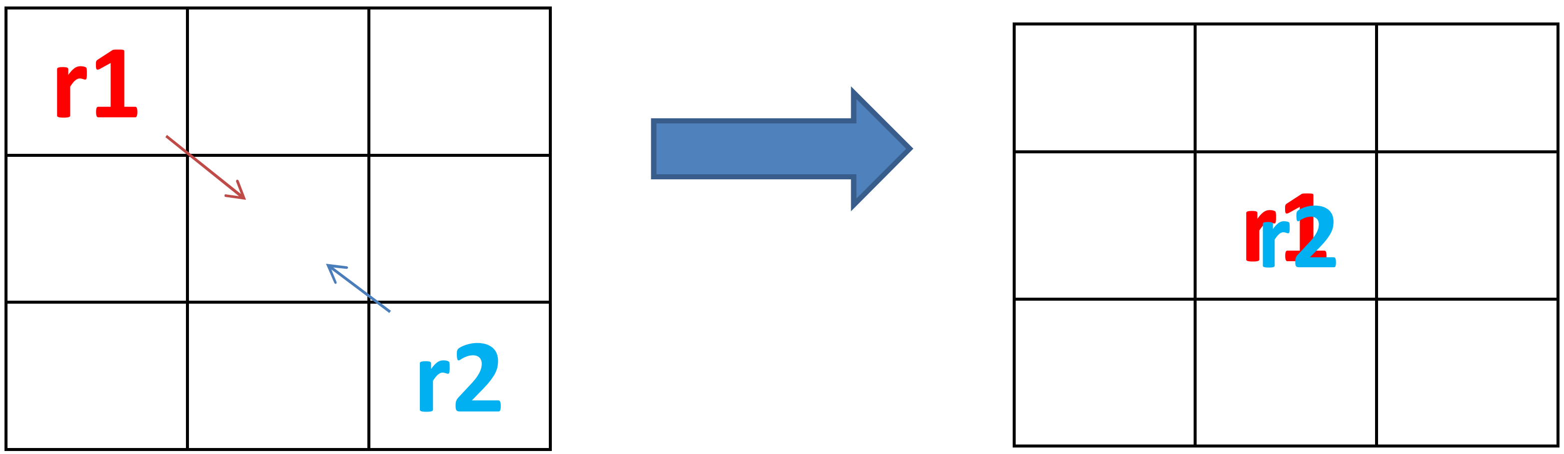
**CS15** Also picture?
Caitlin Sadowski, 7/20/2010

CS17

# Atomicity Violation Example 1: Naïvely Parallelized AntiSocialRobots

- Robots are moved in parallel
  - Check if destination cell free, then move
- This sequence is not atomic!
  - Cell may fill between check and move
  - Schedule of events:

r1.check, r2.check, r1.move, r2.movCRASH



## Slide 6

**CS17** Can we additionally add an even more compelling slide about data race consequences? (either here or at slide 10)

For example, these are virtual robots, but what about a more dangerous example where they were objects in the real world? Could mention Therac-25 disaster, 2003 blackout, etc.

Caitlin Sadowski, 7/14/2010

# Atomicity Violation Example 2: Bank Account

```
int balance = 0;

public void Deposit(int amount)
{
   int b = balance;     // read current balance
   b = b + amount;      // add amount
   balance = b;         // write balance back
}

public void TestParallelDeposit()
{
   Parallel.Invoke(
     () => Deposit(2),
     () => Deposit(5)
   );
   Assert.AreEqual<int>(7, balance);
}
```

- Problematic schedule:
  - task 1 reads balance 0
  - task 2 reads balance 0
  - task 1 writes balance 2
  - task 2 writes balance 5
  - Final balance: 5, not 7!

# Sometimes, it just "looks" atomic… (1)

```
int balance = 0;

public void DepositOne()
{
    balance++;
}

public void TestParallelDepositOne()
{
   Parallel.Invoke(
     () => DepositOne(),
     () => DepositOne()
   );
   Assert.AreEqual<int>(2, balance);
}
```

This is not an atomic operation, because it is internally still executed in three steps:

```
b = balance;
b = b + 1;
balance = b;
```

# Sometimes, it just “looks” atomic... (2)

```
struct RoomPoint {
   public int X;
   public int Y;
   ...
}

RoomPoint p = new RoomPoint(2,3);
r.Location = p;
```

This is not an atomic assignment, because the struct is internally copied in several steps:

r.Location.X = p.X;
r.Location.Y = p.Y;

# Finding Atomicity Problems

- Atomicity Problems are often a result of unexpected concurrency
  - Programmer may not have been aware of issue
- Data races are excellent indicators of potential atomicity problems
  - All of the atomicity problems shown on previous slides are also data races.
- First line of defense against atomicity problems: Prevent data races.

Part 2

# DATA RACES

CS5

# What is a Data Race?

- Two concurrent accesses to a memory location at least one of which is a write.
- Example: Data race between a read and a write

```
int x = 1;
Parallel.Invoke(
                  () => { x = 2; },
                  () => { System.Console.WriteLine(x); }
                );
```

(writes x) (reads x)

- Outcome nondeterministic or worse
  - may print 1 or 2, or arbitrarily bad things on a relaxed memory model

**CS5** Why is it called a data "race"? Can you also explain with a metaphor? Jaeheon has one that he likes involving multiple people trying to get or examine cookies from the same tray.

Caitlin Sadowski, 7/20/2010

# Data Races and Happens-Before

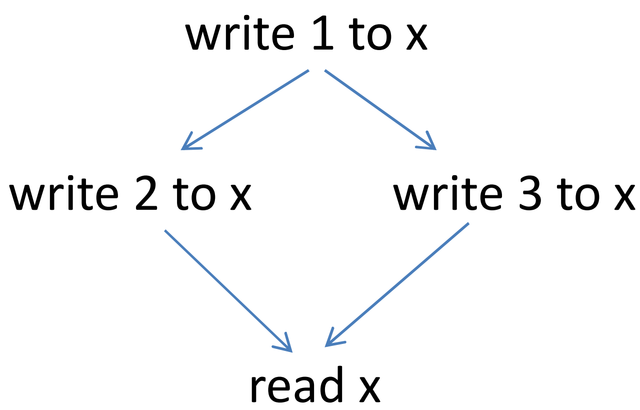
- Example of a data race with two writes:

```
int x = 1;
Parallel.Invoke( () => { x = 2; },
                 () => { x = 3;  }  );
System.Console.WriteLine(x);
```

- We visualize the ordering of memory accesses with a happens-before graph:

There is no path between
(write 2 to x) and (write 3 to x),
thus they are concurrent,
thus they create a data race

(note: the read is not in a data race)



**Slide 13**

**CS4** Another good example to have here would be two threads that both perform (for example) x++, and then ask the class what could happen.
Caitlin Sadowski, 7/19/2010

# Quiz: Where are the data races?

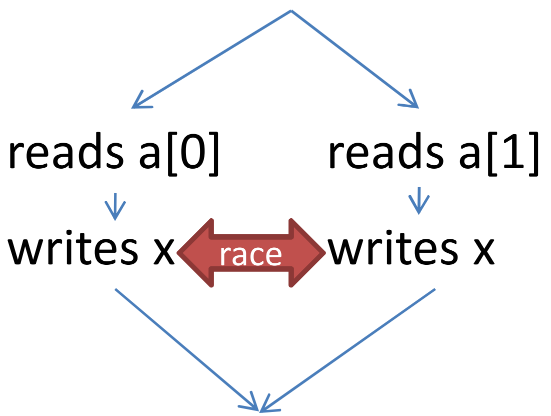
```
Parallel.For(1,2,
i => {
    x = a[i];
});
```

```
Parallel.For(1,2,
i => {
    a[i] = x;
});
```

```
Parallel.For(1,2,
i => {
    a[i] = a[i+1];
});
```

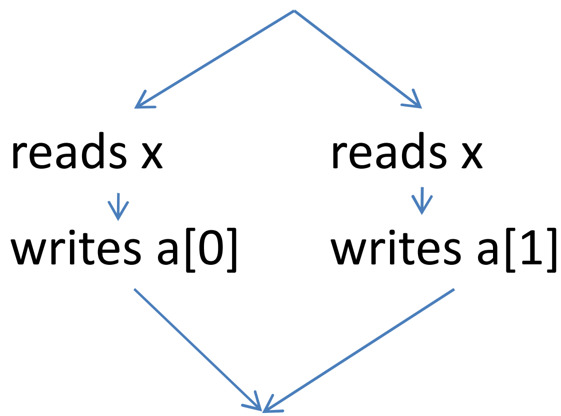
# Quiz: Where are the data races?

```
Parallel.For(1,2,
i => {
     x = a[i];
});
```



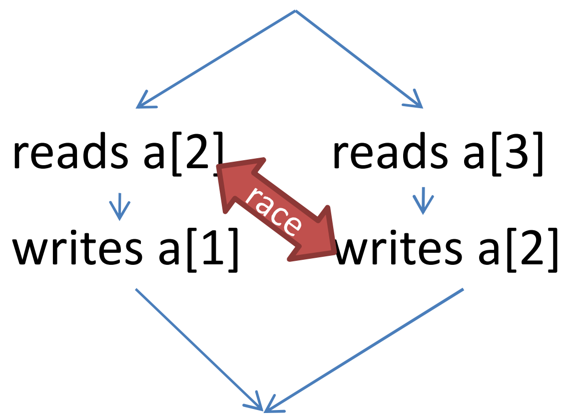

**Race between two writes.**

```
Parallel.For(1,2,
i => {
     a[i] = x;
});
```




**No Race between two reads.**

```
Parallel.For(1,2,
i => {
     a[i] = a[i+1];
});
```




**Race between a read and a write.**

# Spotting Reads & Writes

- Sometimes a single statement performs multiple memory accesses

When you execute

```
x += y
```

there are actually two reads and one write:

reads x
reads y
writes x

When you execute

```
a[i] = x
```

there are actually three reads and one write:

reads x
reads a
reads i
writes a[i]

# Data Races can be hard to spot.

```
Parallel.For(0, 10000,
    i => {a[i] = new Foo();})
```

- Code looks fine... at first.

# Data Races can be hard to spot.

```
Parallel.For(0, 10000,
    i => {a[i] = new Foo();})
```

- Problem: we have to follow calls… even if they look harmless at first (like a constructor).

```
class Foo {
        private static int counter;
        private int unique_id;
        public Foo()
        {
                unique_id = counter++;
        }
}
```

**Data Race on static field !**

# In this Course: We Strictly Follow DRF Discipline

- **Data-Race-Free (DRF) Discipline**

means we avoid ALL data races (no such thing as a “benign” data race).

- Already “best practice” for many, but not all programmers.

# DRF Discipline Advantages

- Avoid issues with memory model
- Make code more declarative
- Make Race Detection Tools More Useful

Race Detectors are excellent at finding

- forgotten dependencies
- unexpected conflicts
- Atomicity problems

Part 3

# DATA RACE PREVENTION

# Avoiding Data Races

- The three most frequent ways to avoid data races on a variable
  - Make it ***isolated***
    - variable is only ever accessed by one task
  - Make it ***immutable***
    - variable is only ever read
  - Make it ***synchronized***
    - Use a lock to arbitrate concurrent accesses
- Can use combination over time

# Labeling memory accesses

- This loop is race-free because the accessed locations are either isolated or immutable!

```
Parallel.For(1,1000,
i => {
    a[i] = x;
});
```

**x**: immutable

**i:** isolated

**a**: immutable

**a[i]**: isolated

# Cool Trick: Avoid Data Races By Encapsulation

```
public class Coordinate
{
   private double x, y;

   public Coordinate(double a,
                     double b)
   {
      x = a;
      y = b;
   }
   public void GetX() {
      return x;
   }
   public void GetY() {
      return y;
   }
}
```

- No data race on x or y
  - *isolated* during construction (no other tasks can access this object yet)
  - *immutable* once constructor is finished (fields x,y are private, and methods only read from them)

Part 4

# BASIC LOCKING

# Using locks

- We can often restrict accesses to a variable and
  - make it ***isolated***
    - variable is only ever accessed by one task
  - make it ***immutable***
    - variable is only ever read
- But if we want to decide on-the-fly who gets to access the variable, we
  - make it ***synchronized***
    - Use a lock to arbitrate between concurrent accesses

# Use lock to arbitrate accesses

- Example: No data race on variable x, because lock guarantees mutual exclusion!

```
int x = 1;
Object mylock = new Object();

Parallel.Invoke(
      () => {   lock(mylock)
                {
                    x = 2;
                }
            },
      () => {   lock(mylock)
                {
                    System.Console.WriteLine(x);
                }
            });
```

writes x

reads x

CS6

# Basic Locking

- Any object can serve as a "lock"
  - At most one task can have the lock at a time

- Task acquires/releases the lock

  C# syntax:

called "critical section"

```
lock(myobj)
{
        . . . code to execute with lock goes here . . .
}
```

- Lock acquired when task enters critical section
  - May have to wait until lock becomes available
- Lock released when task exits critical section
  - When exiting either normally or due to an uncaught exception

**Slide 28**

---

**CS6** You have lock in quotes here; can you also have motivation for this term? (perhaps on a different slide) Specifically, what is the lock metaphor?
Caitlin Sadowski, 7/20/2010

CS7

# Lock Semantics

- Can not enter critical section unless lock is available
  - Task blocks indefinitely if lock is currently held by a different task
  - Several tasks may be blocked & wait for the same lock to become available
  - Tasks may ‘race’ to acquire a lock.. This is not a data race, but a ‘controlled’ race. Winner is chosen nondeterministically.
- Reentrance is o.k.
  - **lock (x) { lock (x) { … } }** is equivalent to **lock(x) { …}**
  - nesting depth is tracked automatically

**Slide 29**

---

**CS7** Can you have a picture here instead of (or in addition to) the text?
Caitlin Sadowski, 7/20/2010

# Using Locks to Prevent Races

- This idiom is commonly used to prevent data races on a field x:
  - Choose some lock to “protect” x
  - Ensure lock held whenever x is accessed

```
Object mylock = new Object(); // use this lock to protect x
Parallel.Invoke(
    () => { lock(mylock) { x = 2; } },
    () => { w = 2; },
    () => { z = 2; },
    () => { lock(mylock) { y = x; } },
)
```

# Often: protect local fields

- Create a private lock object to protect the fields
- Guarantees: no data race on field
  - isolated during construction
  - synchronized afterwards

```
public class SafeCounter
{
   private int value;
   private Object mylock
      = new Object();

   public SafeCounter()
   {
      value = 0;
   }

   public void Increment()
   {
      lock(mylock)
      {
         value = value + 1;
      }
   }
}
```

# Simple Locking Policy

- For each field that may be accessed concurrently:
  - Designate a lock object (in your mind, and with comments)
  - Any object will do (can use *this* as well)
  - The same lock object can be used for many fields
- Every time you access a synchronized object:
  - Make sure the lock is held during the access
- Guarantees: no *data* races

Part 5

# COMMON PROBLEMS WITH LOCKS

# Locks = Easy Fix for all problems ?

- Suppose you have locked everything, and your code is data-race-free.
- Is that the end of all trouble? No.
  - Can still have correctness problems:
    - Deadlocks
    - Atomicity problems
  - Can still have performance problems:
    - Lock contention
    - Locking overhead

CS10

# Pitfall 1: Deadlock

```
Object A = new Object();
Object B = new Object();
Parallel.Invoke(
    () => { lock(A) { lock (B) { ; } } },
    () => { lock(B) { lock (A) { ; } } },
)
```

Deadlocking schedule:

- Task 1 acquires A
- Task 2 acquires B
- Task 1 tries to acquire B, waits for Task 2 to release B
- Task 2 tries to acquire A, waits for Task 1 to release A
- Deadlock! Nobody can make progress

**Slide 35**

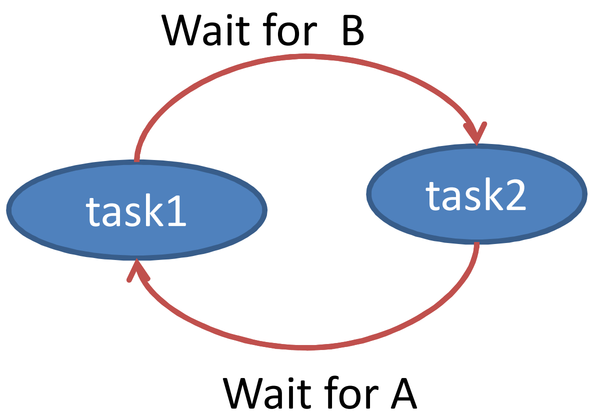
**CS10** Maybe put a more exciting/motivating example here first? I have used one with a lock per bank account, and a trasfer method that involves locking first the "from" account and then the "to" account.

Caitlin Sadowski, 7/20/2010

# Deadlock = Cycle in Wait-For Graph

- Visualize deadlocked configuration
  - Each vertex represents a waiting task
  - Each edge x->y represents "x is waiting for y"

```
Parallel.Invoke(
    () => { lock(A) { lock (B) { ; } } },
    () => { lock(B) { lock (A) { ; } } },
)
```




- If there is a cycle: potential deadlock
- If there is no cycle: safe

# Solution: Consistent Order

- If locks are acquired in consistent order, no cycle and thus no deadlock!

```
Object A = new Object();
Object B = new Object();
Object C = new Object();

Parallel.Invoke(
    () => { lock(A) { lock (B) { ; } } },
    () => { lock(B) { lock (C) { ; } } },
    () => { lock(A) { lock (C) { ; } } },
)
```

# Lock Leveling

- Simple policy to avoid deadlocks:
  - Assign a "level" (some arbitrarily chosen number) to each lock (in your mind and in comments).
  - Follow policy: whenever acquiring a lock, its level must be higher than all the levels of the locks already held
  - Effect: no cycles possible

# Pitfall 2: Atomicity Violation

- Consider AntiSocial Robots Code:

```
Parallel.Invoke(
    () => { if (IsFree(r1.Destination)) MoveRobot(r1);  },
    () => { if (IsFree(r2.Destination)) MoveRobot(r2); }
)
Bool IsFree(Cell c)  {
   lock(c) { return c.Robot == null };
}
Void MoveRobot(Robot r) {
   lock(r.Destination)  { r.Destination.Robot = r; }
}
```

- Problem: robots may move into same cell!
  - Because lock is released & re-acquired in between checking & moving!

# Next Lecture

- Typical Performance Problems
- Detailed Case Study: Antisocial Robots

# http://code.msdn.microsoft.com/ParExtSamples

- ParallelExtensionsExtras.
  - FastBitmap

# Parallel Programming with Microsoft .NET

- Appendix B (Debugging and Profiling Parallel Applications)