Microsoft® Research
Faculty Summit 2010

A New Approach to Concurrency and Parallelism (Part 2)

Tom Ball (tball@microsoft.com)
Sebastian Burckhardt (sburckha@microsoft.com)
Madan Musuvathi (madanm@microsoft.com)
Microsoft Research

Practical Parallel and Concurrent Programming (PP&CP)

 **P&C**	**Parallelism**	**Concurrency**
Performance	Speedup	Responsiveness
Correctness	Atomicity, Determinism, Deadlock, Livelock, Linearizability, Data races, …	

PPCP Courseware: http://research.microsoft.com/ppcp/

- **What**: 16 weeks (8 units) of material
  - Slides
  - Notes
  - Exercises, quizzes
  - Sample programs and applications
  - Tests and tools
- **Who**: beginning graduates, senior undergraduates
- **Prerequisites**: OO programming, systems, data structures
- **Dependencies**:
  - .NET 4
  - C# and F# languages

PPCP Units: Breadth with Correctness Concepts

- Unit 1: Imperative Data Parallel Programming
- Unit 2: Shared Memory
- Unit 3: Concurrent Components
- Unit 4: Functional Data Parallel Programming
- Unit 5: Scheduling and Synchronization
- Unit 6: Interactive/Reactive Systems
- Unit 7: Message Passing
- Unit 8: Advanced Topics

CHESS Concurrency Testing Technology

http://research.microsoft.com/chess/

- Source code release
  - chesstool.codeplex.com
- Preemption bounding [PLDI07]
  - speed search for bugs
  - simple counterexamples
- Fair stateless exploration [PLDI08]
  - scales to large programs
- Architecture [OSDI08]
  - Tasks and SyncVars
  - API wrappers
- LineUp: automatic linearizability checking [PLDI10]
- Data race detection
- Memory model issues
- Coming:
  - Concurrency unit tests
  - Determinism checking

CHESS Tested

Some Correctness Concepts Featured in PPCP

- Data race free discipline and happens-before data race detection
- Automated linearizability checking of concurrent components
- Supported by CHESS

Microsoft® Research
Faculty Summit 2010

Data Races

Data Race Free (DRF) Discipline
Happens-Before Race Detection

Why Care About Data Races?

- Data races may reveal synchronization errors
  - Many errors (from simple omissions to algorithmic mistakes) can manifest as data races.
  - Data race detectors can often help to find & fix concurrency bugs very efficiently.
  - But: some data races may appear “benign”, watering down the utility of such detectors (false alarms)

- Data races are not portable
  - Behavior of program with data races depends on memory model
  - Relaxations in compiler or hardware may introduce strange & platform-dependent effects

What is a Data Race, Traditionally?

- Long history, many definitions
- Sometimes linked to specific programming idioms
  - “shared variables must be lock-protected”
- Often unclear terminology
  - “Races” vs. “Data Races”: Is it a race if two threads try to acquire the same lock?
  - “Ordered by synchronization”: What counts as synchronization?
- Recently: *Convergence of Definition*
  - Motivated by research on memory models and recent proposals for language-level memory models (Java, C++)

What is a Data Race, Today/Tomorrow?

- If two *conflicting* memory accesses happen *concurrently*, we have a data race.
- Two memory accesses *conflict* if
    - They target the same location
    - They are not both reads
    - They are not both synchronization operations

Proposal: Follow DRF Discipline

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

  means we write programs that have NO data races (not even "benign" ones).

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

How Does DRF Discipline Affect my Programs ?

- ~~Answer A:~~
  ~~I have to protect everything with locks and must not use lock-free synchronization techniques~~
- Answer B:
  I have to properly declare racy accesses using type qualifiers (atomic, volatile) or special operations (interlocked, compare-and-swap)

DRF Discipline Pros & Cons

- Pros
  - Code is more declarative (easier to see intentions)
  - Code is immune against memory model relaxations (= why DRF invented in the first place).
  - All data races are bugs, no benign races.
  - Code is easier to verify and debug.
- Cons
  - Have to learn how to use type qualifiers correctly
  - Annotation overhead (not much)
  - Some qualifiers not efficient on some platforms

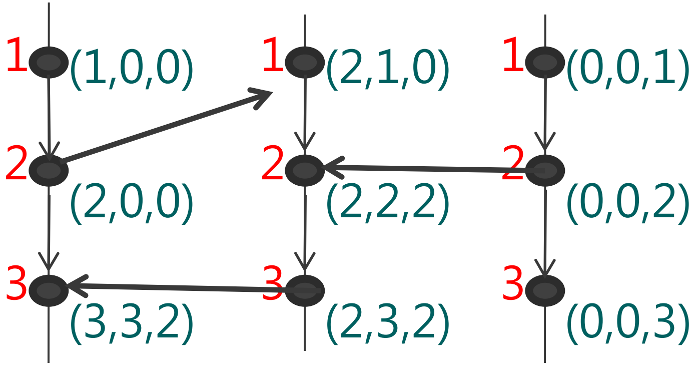
How to find a data race?

- Test for concurrent conflicting accesses
  - Problem: schedule varies from run to run
  - Probability of making potentially concurrent accesses actually simultaneous often not very good.

- Idea: happens-before race detector
  - Check for conflicting accesses that could have been concurrent in a slightly different schedule

Happens-Before Order [Lamport]

- Use logical clocks and timestamps to define a partial order called *happens-before* on events in a concurrent system
- States *precisely* when two events are *logically* concurrent (abstracting away real time)



- Cross-edges from send events to receive events
- (a_1, a_2, a_3) happens before (b_1, b_2, b_3) iff $a_1 \leq b_1$ and $a_2 \leq b_2$ and $a_3 \leq b_3$

Happens-Before for Shared Memory

- Distributed Systems
  Cross-edges from send to receive events
- Shared Memory systems
  Cross-edges represent ordering effect of synchronization
  - Edges from lock release to subsequent lock acquire
  - Edges from volatile writes to subsequent volatile reads
  - Long list of primitives that may create edges
    - Semaphores, Waithandles, Rendezvous, system calls (asynchronous IO), …

Example

Static Program

```
int data;
volatile bool flag;
```

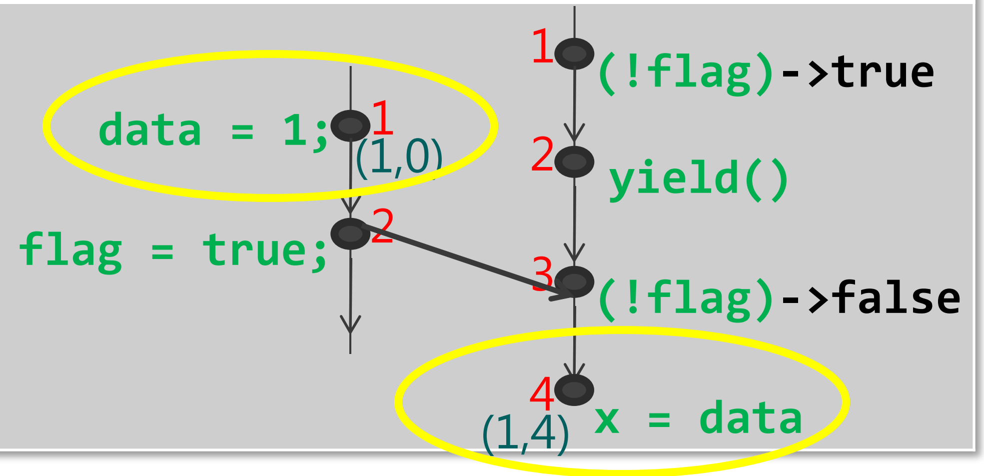
Thread 1

```
data = 1;
flag = true;
```

Thread 2

```
while (!flag)
    yield();
int x = data;
```

Dynamic Execution Trace



- Not a data race because (1,0) ≤ (1,4)
- If flag were not declared volatile, we would not add a cross-edge, and this would be a data race.

Microsoft® Research
Faculty Summit 2010

Automated Linearizability Checking

Madan Musuvathi
Microsoft Research

Joint work with
Sebastian Burckhardt, MSR
Chris Dern, MS
Roy Tan, MS

An Implementation of Concurrent Queue

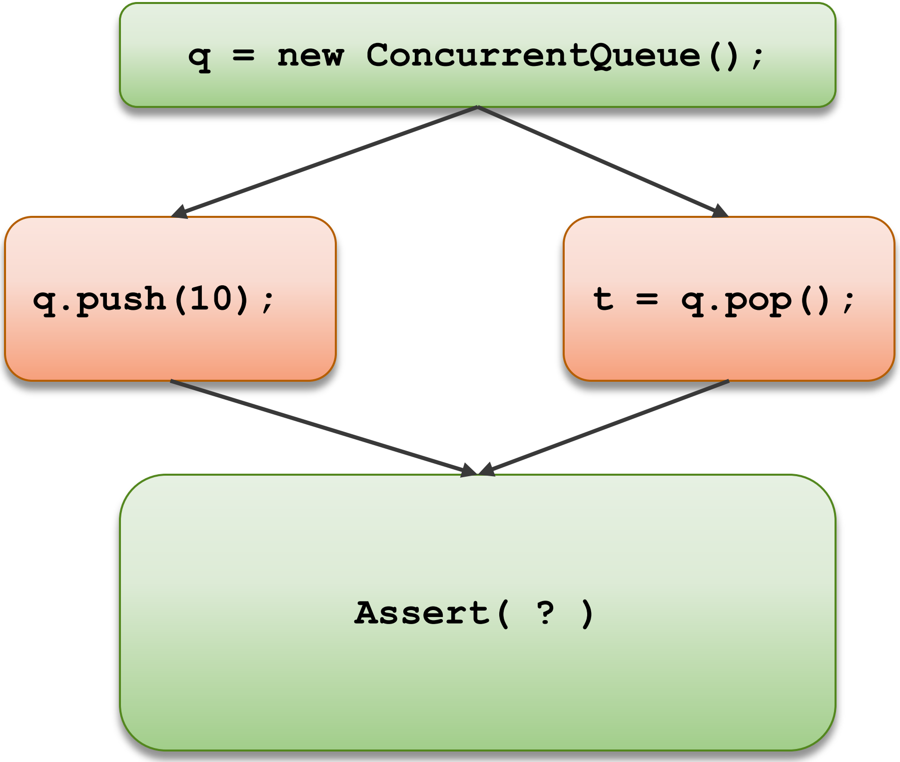
```
#pragma warning disable 0420

// ==++==
//
//   Copyright (c) Microsoft Corporation.  All rights reserved.
//
// ==--==
//
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
=+=+=+=+=+
//
// ConcurrentQueue.cs
//
// <OWNER>csong</OWNER>
//
// A lock-free, concurrent queue primitive, and its associated debugger view type.
//
// =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using BadSystemDiagnosticsContracts;
using System.Runtime.ConstrainedExecution;
using System.Runtime.InteropServices;
using System.Runtime.Serialization;
using System.Security;
using System.Security.Permissions;
using BadSystemThreading;
using BadSystem;
using System.Threading;

namespace BadSystemCollectionsConcurrent
{

    /// <summary>
    /// Represents a thread-safe first-in, first-out collection of objects.
    /// </summary>
    /// <typeparam name="T">Specifies the type of elements in the queue.</typeparam>
    /// <remarks>
    /// All public  and protected members of <see cref="ConcurrentQueue{T}"/> are thread-safe and may be used
    /// concurrently from multiple threads.
    /// </remarks>
    [ComVisible(false)]
    [DebuggerDisplay("Count = {Count}")]
    [DebuggerTypeProxy(typeof(SystemCollectionsConcurrent_ProducerConsumerCollectionDebugView<>))]
    [HostProtection(Synchronization = true, ExternalThreading = true)]
    [Serializable]
    public class ConcurrentQueue<T> : IProducerConsumerCollection<T>
    {
        //fields of ConcurrentQueue
        [NonSerialized]
        private volatile Segment m_head;

        [NonSerialized]
        private volatile Segment m_tail;

        private T[] m_serializationArray; // Used for custom serialization.

        private const int SEGMENT_SIZE = 32;
```

```
/// <summary>
        /// Get the data array to be serialized
        /// </summary>
        [OnSerializing]
        private void OnSerializing(StreamingContext context)
        {
            // save the data into the serialization array to be saved
            m_serializationArray = ToArray();

        /// <summary>
        /// Construct the queue from a previously seiralized one
        /// </summary>
        [OnDeserialized]
        private void OnDeserialized(StreamingContext context)
        {
            Contract.Assert(m_serializationArray != null);
            InitializeFromCollection(m_serializationArray);
            m_serializationArray = null;
        }

        /// <summary>
        /// Copies the elements of the <see cref="T:System.Collections.ICollection"/> to an <see
        /// cref="T:System.Array"/>, starting at a particular
        /// <see cref="T:System.Array"/> index.
        /// </summary>
        /// <param name="array">The one-dimensional <see cref="T:System.Array">Array</see> that is the
        /// destination of the elements copied from the
        /// <see cref="T:System.Collections.Concurrent.ConcurrentBag"/>. The <see
        /// cref="T:System.Array">Array</see> must have zero-based indexing.</param>
        /// <param name="index">The zero-based index in <paramref name="array"/> at which copying
        /// begins.</param>
        /// <exception cref="ArgumentNullException"><paramref name="array"/> is a null reference (Nothing in
        /// Visual Basic).</exception>
        /// <exception cref="ArgumentOutOfRangeException"><paramref name="index"/> is less than
        /// zero.</exception>
        /// <exception cref="ArgumentException">
        /// <paramref name="array"/> is multidimensional. -or-
        /// <paramref name="array"/> does not have zero-based indexing. -or-
        /// <paramref name="index"/> is equal to or greater than the length of the <paramref name="array"/>
        /// -or- The number of elements in the source <see cref="T:System.Collections.ICollection"/> is
        /// greater than the available space from <paramref name="index"/> to the end of the destination
        /// <paramref name="array"/>. -or- The type of the source <see
        /// cref="T:System.Collections.ICollection"/> cannot be cast automatically to the type of the
        /// destination <paramref name="array"/>.
        /// </exception>
        void ICollection.CopyTo(Array array, int index)
        {
            // Validate arguments.
            if (array == null)
            {
                throw new ArgumentNullException("array");
            }

            // We must be careful not to corrupt the array, so we will first accumulate an
            // internal list of elements that we will then copy to the array. This requires
            // some extra allocation, but is necessary since we don't know up front whether
            // the array is sufficiently large to hold the stack's contents.
            ((ICollection)ToList()).CopyTo(array, index);
        }
```

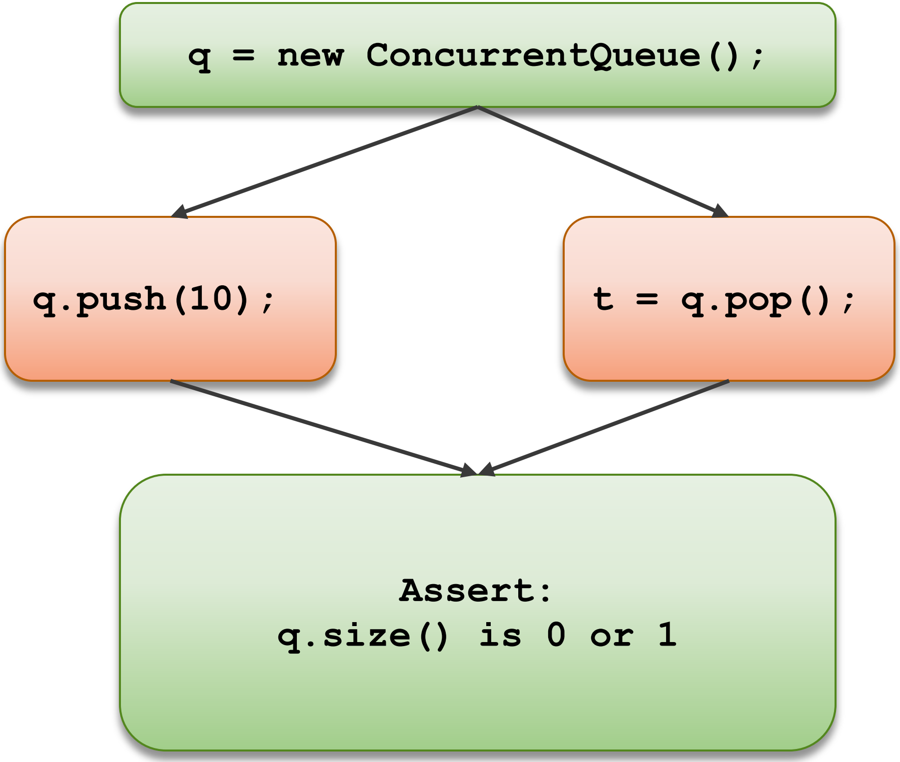
```
public bool IsEmpty
        {
            get
            {
                Segment head = m_head;
                if (!head.IsEmpty)
                    //fast route 1:
                    //if current head is not empty, then queue is not empty
                    return false;
                else if (head.Next == null)
                    //fast route 2:
                    //if current head is empty and it's the last segment
                    //then queue is empty
                    return true;
                else
                //slow route:
                //current head is empty and it is NOT the last segment,
                //it means another thread is growing new segment
                {
                    SpinWait spin = new SpinWait();
                    while (head.IsEmpty)
                    {
                        if (head.Next == null)
                            return true;

                        spin.SpinOnce();
                        head = m_head;
                    }
                    return false;
                }
            }
        }

        /// <summary>
        /// Copies the elements stored in the <see cref="ConcurrentQueue{T}"/> to a new array.
        /// </summary>
        /// <returns>A new array containing a snapshot of elements copied from the <see
        /// cref="ConcurrentQueue{T}"/>.</returns>
        public T[] ToArray()
        {
            return ToList().ToArray();
        }

        /// <summary>
        /// Copies the <see cref="ConcurrentQueue{T}"/> elements to a new <see
        /// cref="T:System.Collections.Generic.List{T}"/>.
        /// </summary>
        /// <returns>A new <see cref="T:System.Collections.Generic.List{T}"/> containing a snapshot of
        /// elements copied from the <see cref="ConcurrentQueue{T}"/>.</returns>
        private List<T> ToList()
        {
            //store head and tail positions in buffer,
            Segment head, tail;
            int headLow, tailHigh;
            GetHeadTailPositions(out head, out tail, out headLow, out tailHigh);

            if (head == tail)
            {
                return head.ToList(headLow, tailHigh);
            }
```

• • •

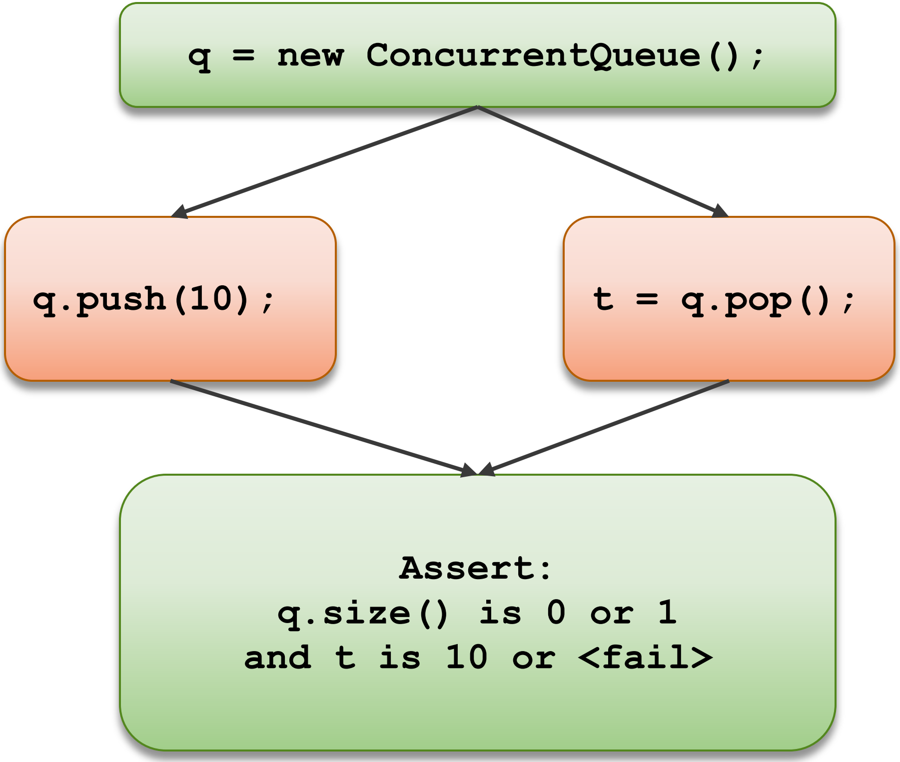
Let's write a test



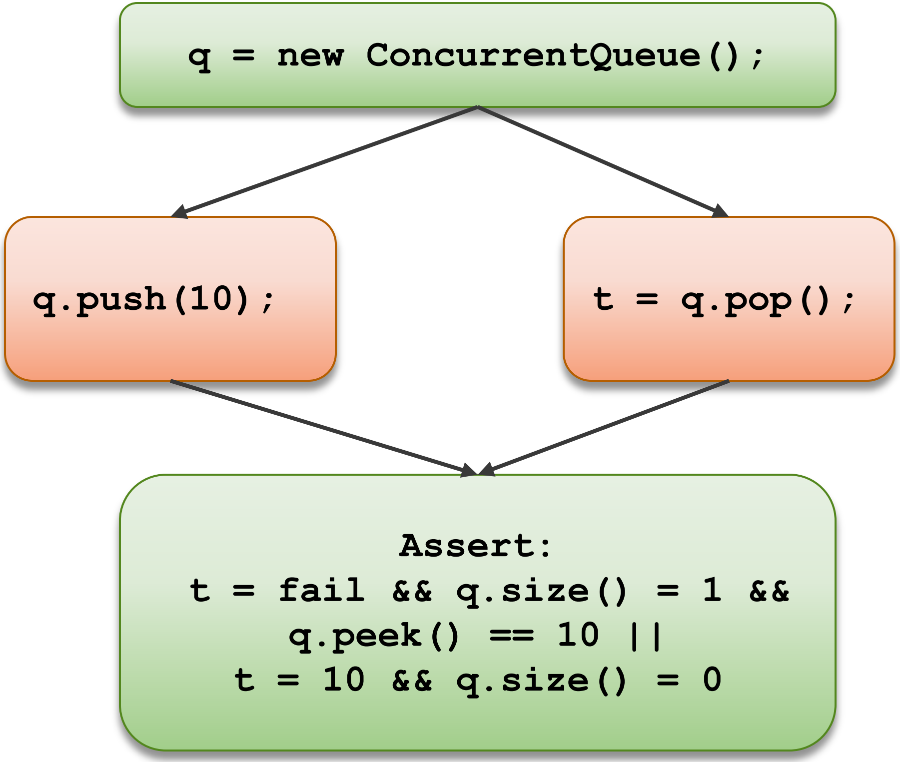
Let's write a test



Let's write a test



Let's write a test



Let's write a test

```
q = new ConcurrentQueue();
```

```
q.push(10);
t = q.pop();
```

```
q.push(20);
u = q.pop();
```

```
Assert ( ? )
```

Let's write a test

```
q = new ConcurrentQueue();
```

```
q.push(10);
t = q.pop();
```

```
q.push(20);
u = q.pop();
```

```
Assert:
q.size() == 0 &&
(t = 10 || t = 20) &&
(u = 10 || t = 20) &&
u != t
```

Let's write a test

```
q = new ConcurrentQueue();
```

```
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
```

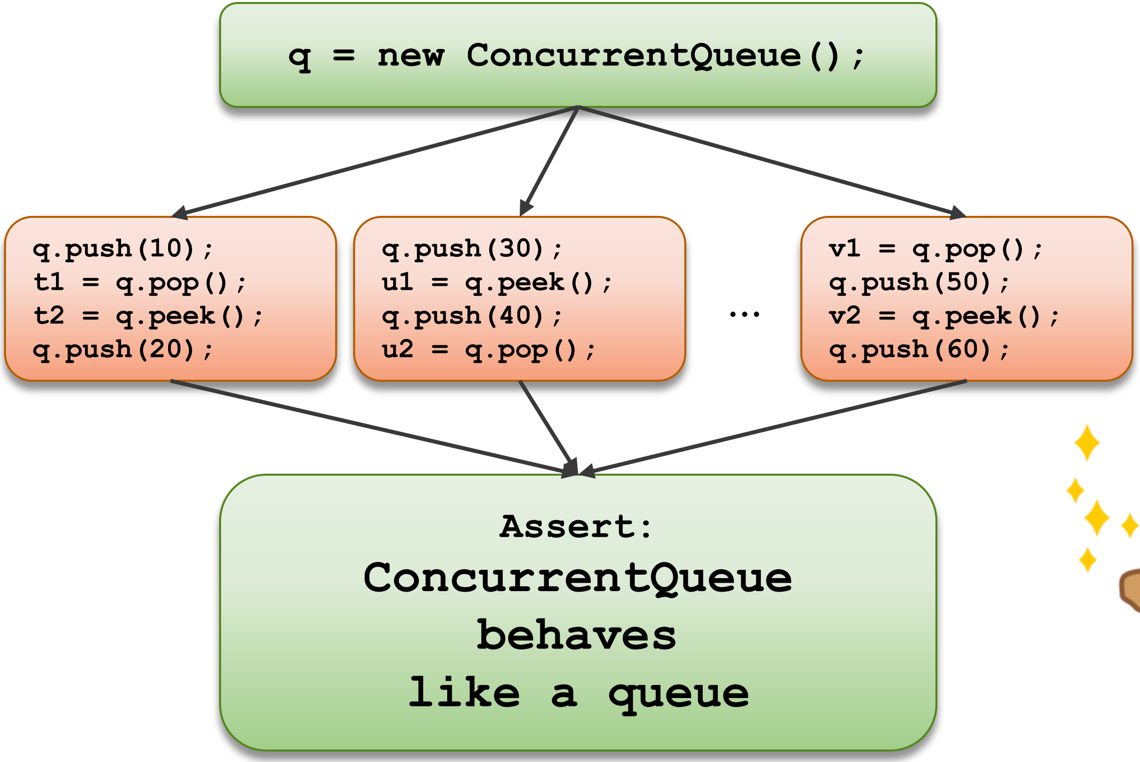
```
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
```

...

```
v1 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);
```

```
Assert ( ? )
```

Wouldn't it be nice if we could just say...



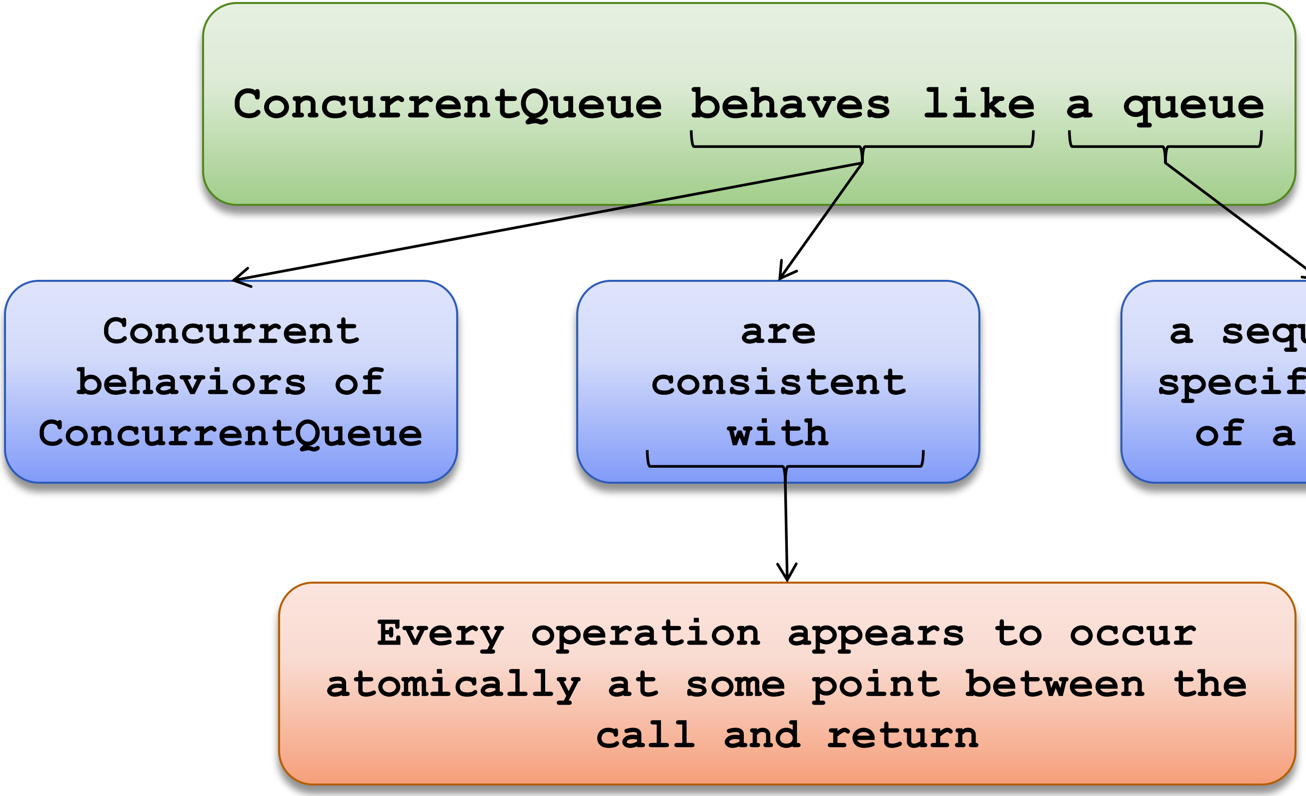
Informally, this is “thread safety”

`ConcurrentQueue behaves like a queue`

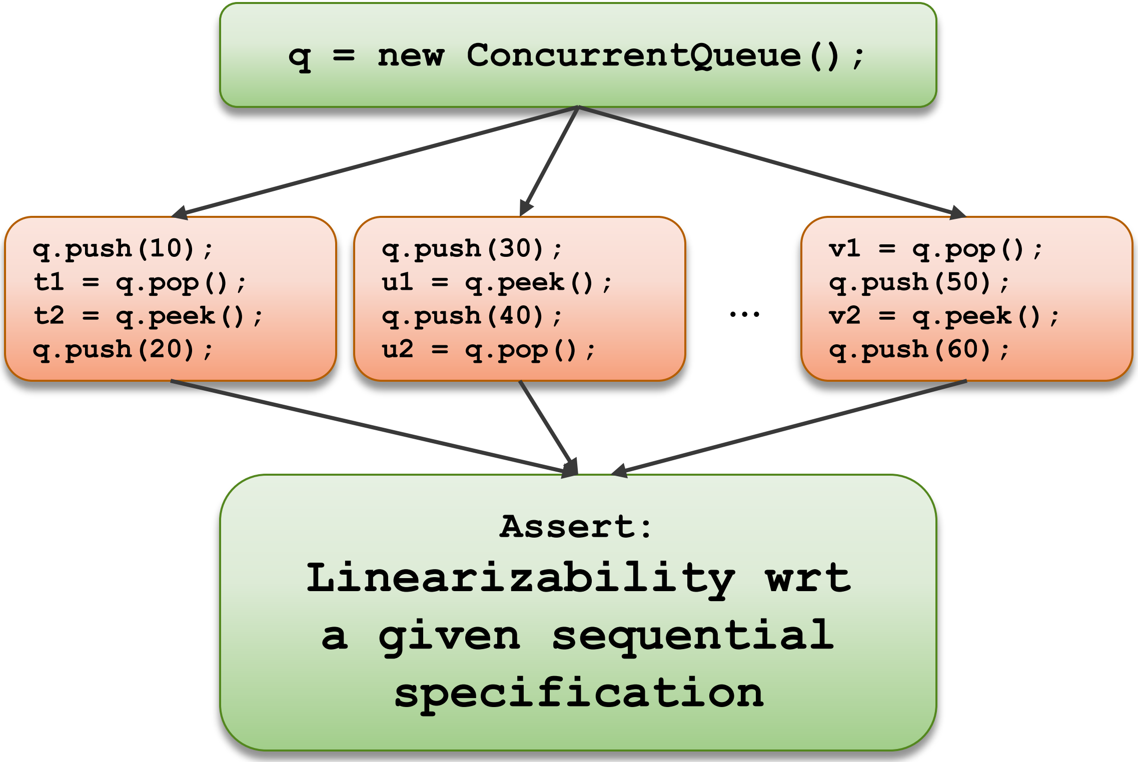
WIKIPEDIA
The Free Encyclopedia

A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads.

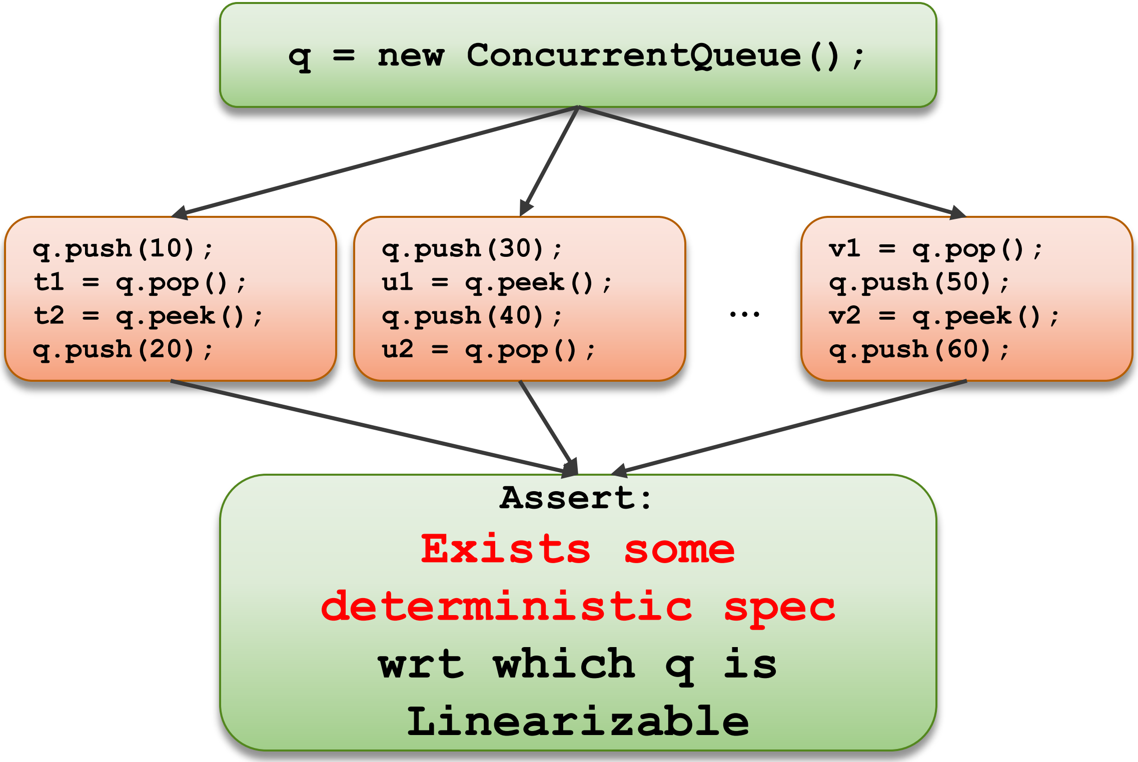
Formally, this is Linearizability [Herlihy & Wing '90]



So, simply check linearizability



LineUp: No need to provide a sequential specification



LineUp Details [see PLDI '10 paper]

- Automatically synthesize a sequential specification
  - By observing sequential behaviors of a component
- Check linearizability with respect to this spec
- Completeness
  - LineUp failure → Component is not linearizable wrt any deterministic spec
- Restricted Soundness
  - Component is not linearizable → Exists a test case for which LineUp fails

Formalizing "Thread Safety"

- Thread safety == Generalized linearizability
- Linearizability does not check against incorrect blocking
  - An implementation that blocks on all operations is vacuously linearizable

Practical Parallel and Concurrent Programming (PP&CP)

 P&C	Parallelism	Concurrency
Performance	Speedup	Responsiveness
Correctness	Atomicity, Determinism, Deadlock, Livelock, Linearizability, Data races, …	

Microsoft® Research
Faculty Summit 2010

Microsoft®