Multicore Programming Pitfalls and Solutions

Madan Musuvathi

Research in Software Engineering

Microsoft Research

My Team at MSR

- Research in Software Engineering RiSE
  - http://research.microsoft.com/rise

My Team at MSR

- Research in Software Engineering RiSE
  - http://research.microsoft.com/rise

- Build program analysis tools
- Research new languages and runtimes
- Study new logics and build faster inference engines
- Improve software engineering methods

http://research.microsoft.com/rise

Research in Software Engineering (RiSE)

RiSE coordinates Microsoft's Research in Software Engineering in Redmond, USA. Our mission is to advance the state of the art in SE, to bring those advances to Microsoft's business, and to take care of those SE technologies that are critical to the company, but not inherently linked to particular products.

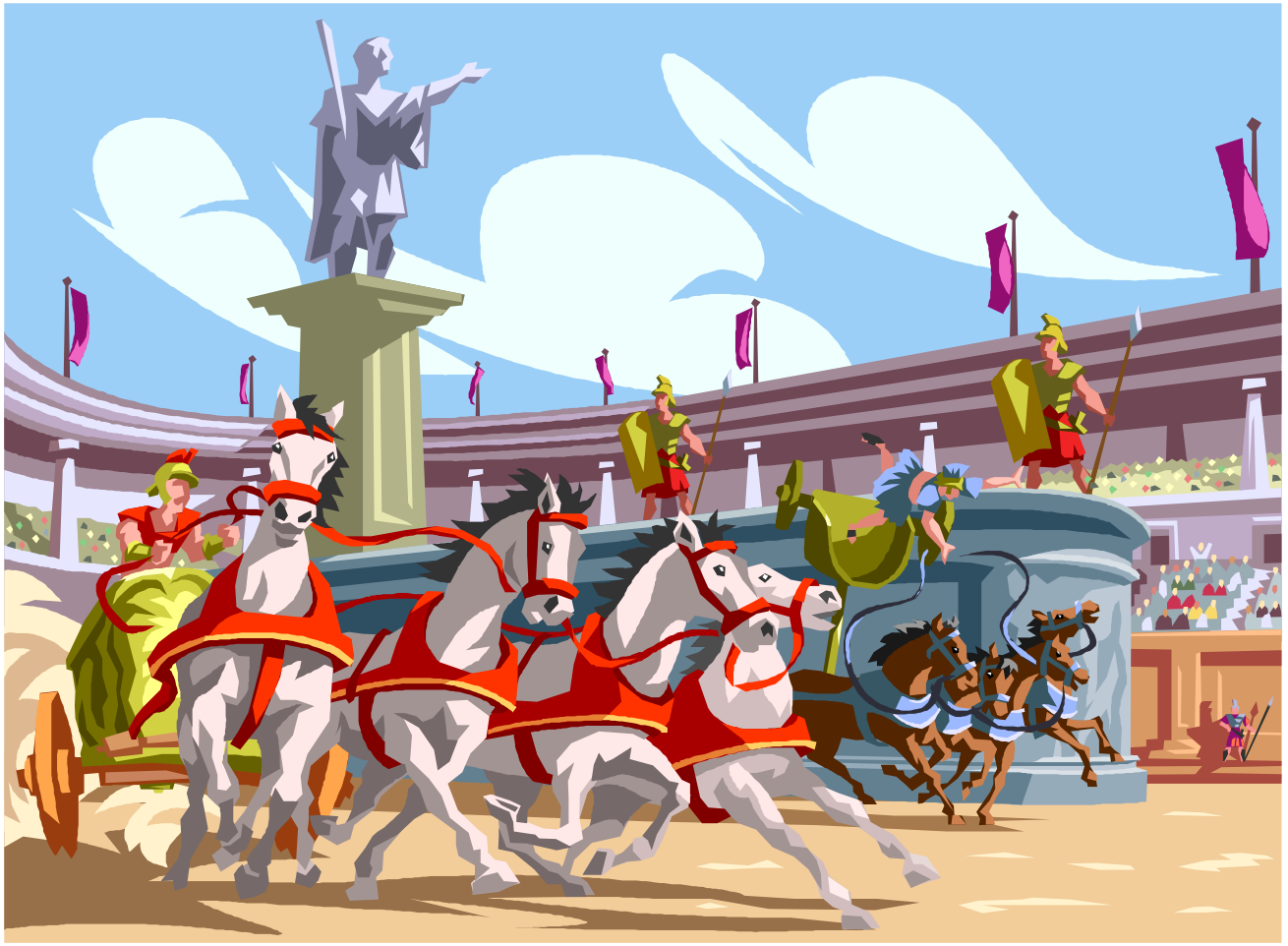
- Tools
- Publications
- People
- Movies on Channel9
- Connect on Facebook
- Past Projects

Madan Musuvathi and Sebastian Burckhardt - Concurrency Fuzzing with Cuzz

see all...

This talk is about

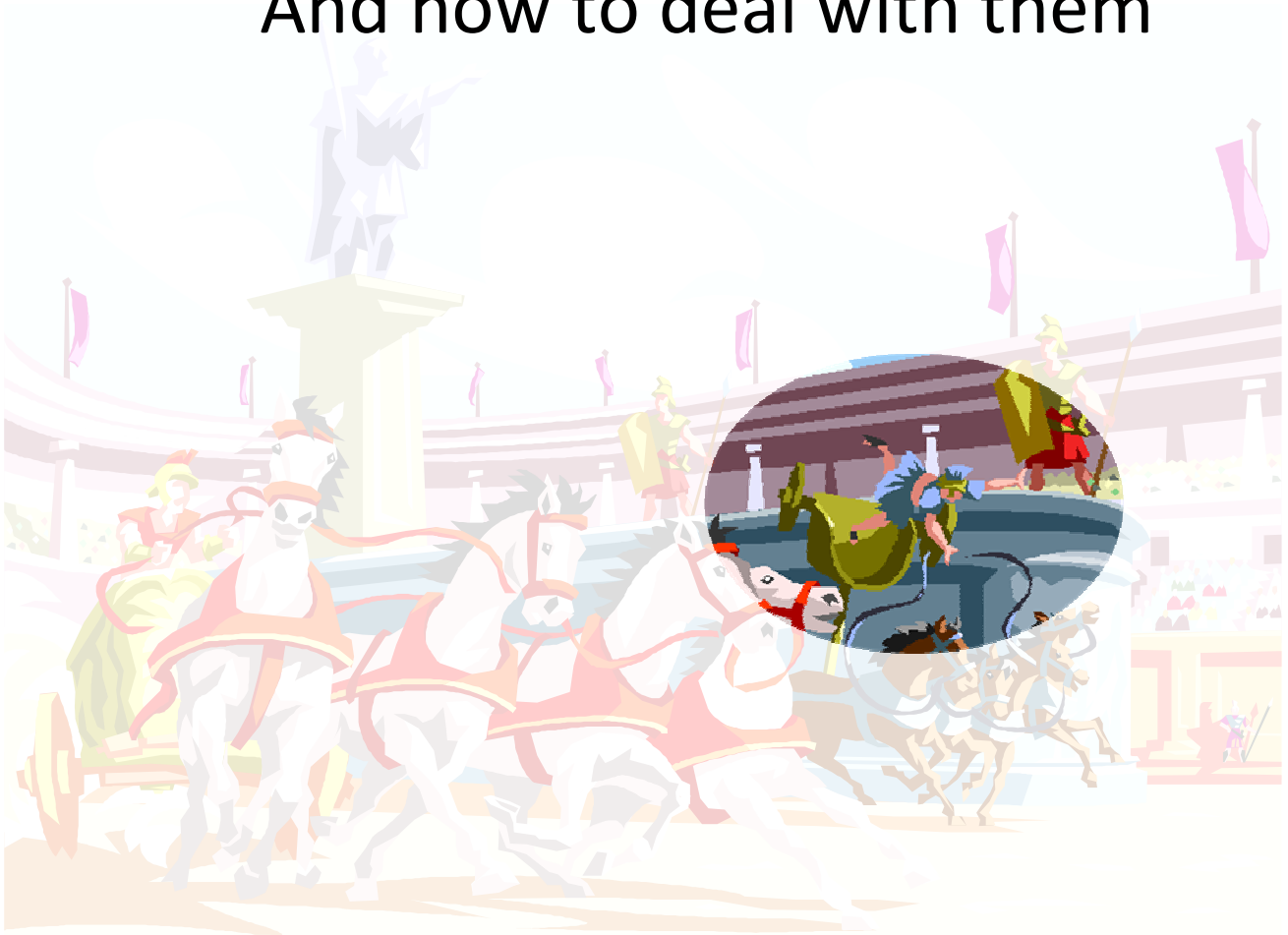
Race Conditions



This talk is about

Race Conditions

And how to deal with them



Race Conditions == Timing Errors

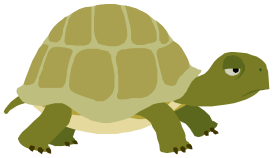
- Unexpected timing of program events
  - Threads, inputs, event handlers, timers, …
- Can result in bugs that are
  - HARD to find
  - HARD to reproduce
  - HARD to debug
  - HARD to fix
- Some of the most expensive in terms of dev/test time

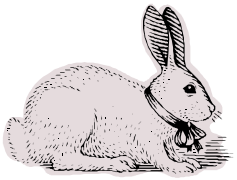
Fundamental Problem

- We don’t know how to make computers do multiple things at the same time

Recent Hardware Trends

- Moore's law = No. of transistors doubles every two years





Moore's law used
to make computers faster

Moore's law now
produces more cores

More programmers exposed
to race conditions

Race Condition Example

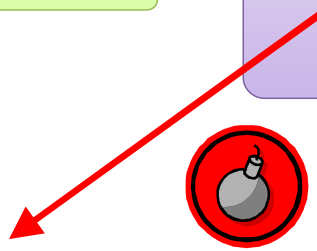
Parent

```
void* p = 0;
CreateThd(child);
```

```
p = malloc(…);
```

Child

```
Init();
DoMoreWork();
p->f ++;
```



Under rare conditions, parent might take a long time to start

Even “single-threaded” programs have race conditions

JavaScript: lingua franca of the web

How to Deal with Race Conditions

Simple Answer

- Identify the sources of nondeterminism
- Identify the space of all behaviors of your program
- Use tools to explore this space

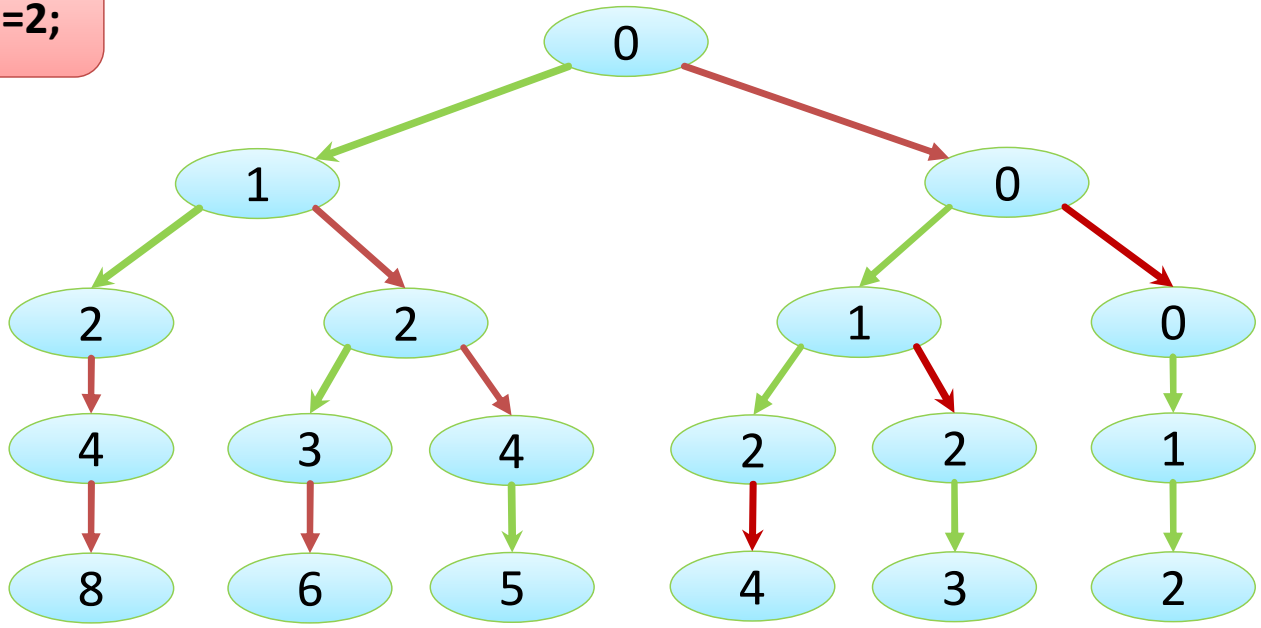
Thread Schedule Space

Thread 1

```
x++;
x++;
```

Thread 2

```
x*=2;
x*=2;
```



Sources of Nondeterminism

- User inputs
- Network Events
- Timers
- Thread interleavings
- System call return values

Cuzz: Concurrency Fuzzing

Cuzz: Concurrency Fuzzing

- Disciplined randomization of thread schedules
- Finds all concurrency bugs in every run of the program
  - With reasonably-large probability
- Scalable
  - In the no. of threads and program size
- Effective
  - Bugs in IE, Firefox, Office Communicator, Outlook, …
  - Bugs found in the first few runs

Concurrency Fuzzing in Three Steps

Parent

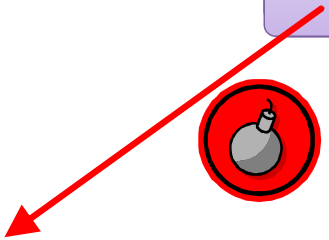
```
void* p = 0;
RandDelay();
CreateThd(child);
```

```
RandDelay();
p = malloc(…);
```

Child

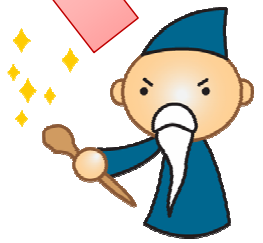
```
Init();
RandDelay();
DoMoreWork();
```

```
RandDelay();
p->f ++;
```



1. Instrument calls to Cuzz
2. Insert random delays
3. Use the Cuzz algorithm to determine when and by how much to delay

This is where all the magic is



Cuzz Demo

Cuzz Algorithm

Inputs: n: estimated bound on the number of threads
k: estimated bound on the number of steps
d: target bug depth

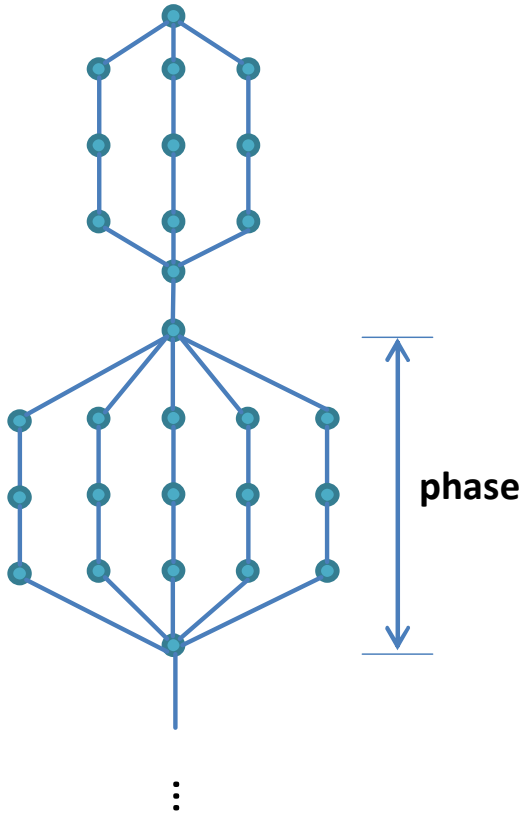
```
// 1. assign random priorities >= d to threads
for t in [1…n]  do  priority[t] = rand() + d;

// 2. chose d-1 lowering points at random
for i in [1...d)  do  lowering[i] = rand() % k;

steps = 0;
while (some thread enabled) {
    // 3. Honor thread priorities
    Let t be the highest-priority enabled thread;
    schedule t for one step;
    steps ++;

    // 4. At the ith lowering point, set the priority to i
     if steps == lowering[i] for some i
         priority[t] = i;
}
```

Probabilistic Guarantee

- Prob (finding a bug)

$$\geq \frac{1}{n.\,k^{d-1}}$$

- n: max no. of concurrent threads
- k: max no. of sync ops per phase
- d: depth of the bug
  - Determines the "complexity" of the bug

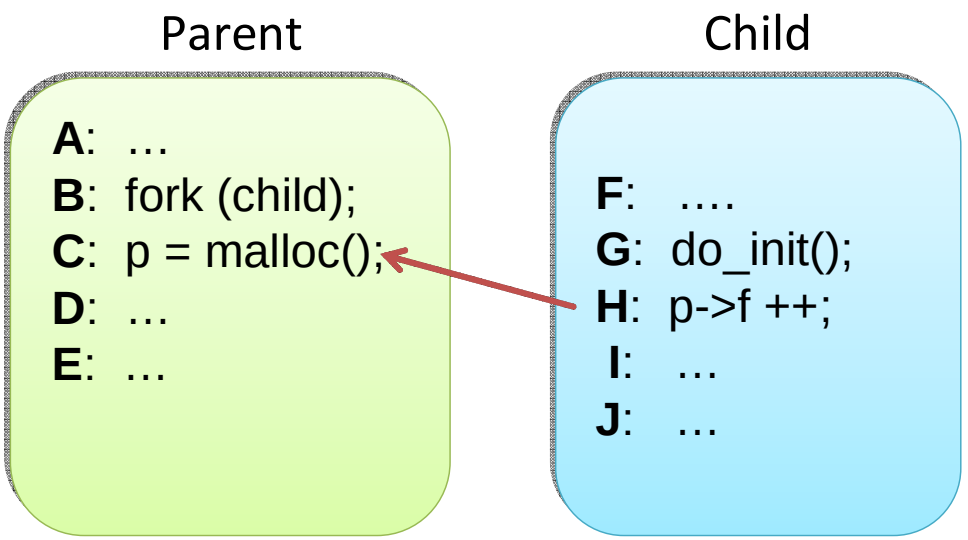


Bug Depth

- Bug depth = number of ordering constraints sufficient to find the bug
- Bugs of higher depth
  - Have a more complex root cause
  - Cuzz finds them with lower probabilistic bounds
- Best explained through examples

A Bug of Depth 1

- Bug Depth = no. of ordering constraints sufficient to find the bug



Possible schedules

A B C D E F G H I J ✓

A B F G H C D E I J ✗

A B F G C D E H I J ✓

A B F G C H D E I J ✓

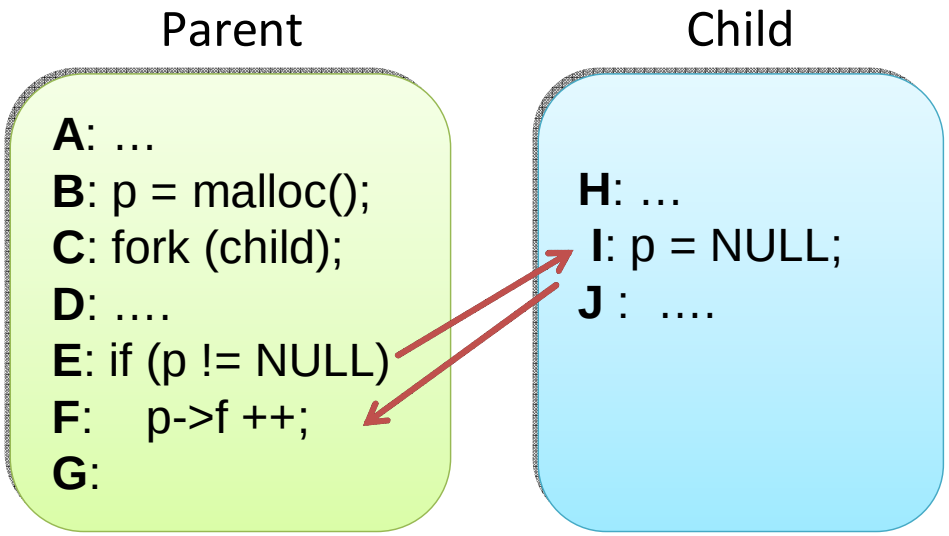
A B F G H I J C D E ✗

…

- Probability of bug >= 1/n
  - n: no. of threads (~ tens)

A Bug of Depth 2

- Bug Depth = no. of ordering constraints sufficient to find the bug



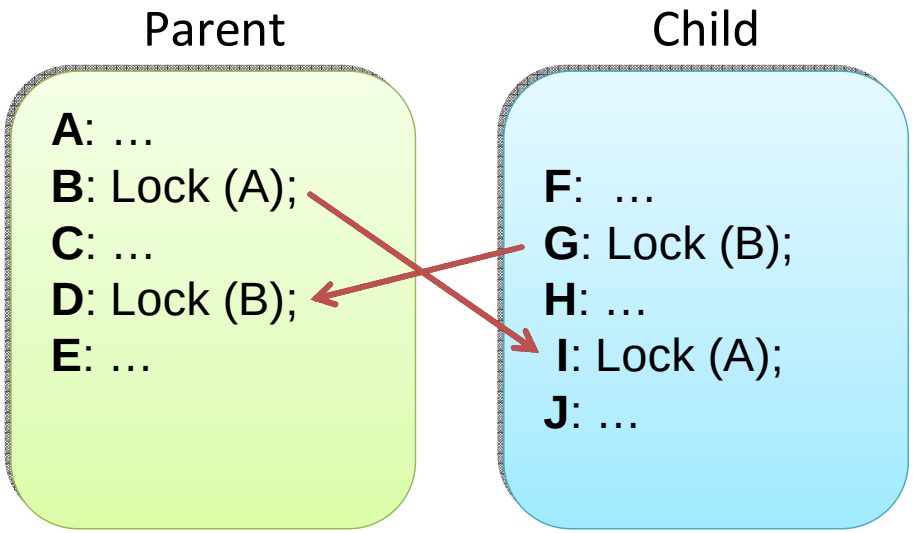
Possible schedules

A B C D E F G H I J ✓
A B C D E H I J F G ✗
A B C H I D E G J ✓
A B C D H E F I J G ✓
A B C H D E I J F G ✗
…

- Probability of bug >= 1/nk
  - n: no. of threads (~ tens)
  - k: no. of instructions (~ millions)

Another Bug of Depth 2

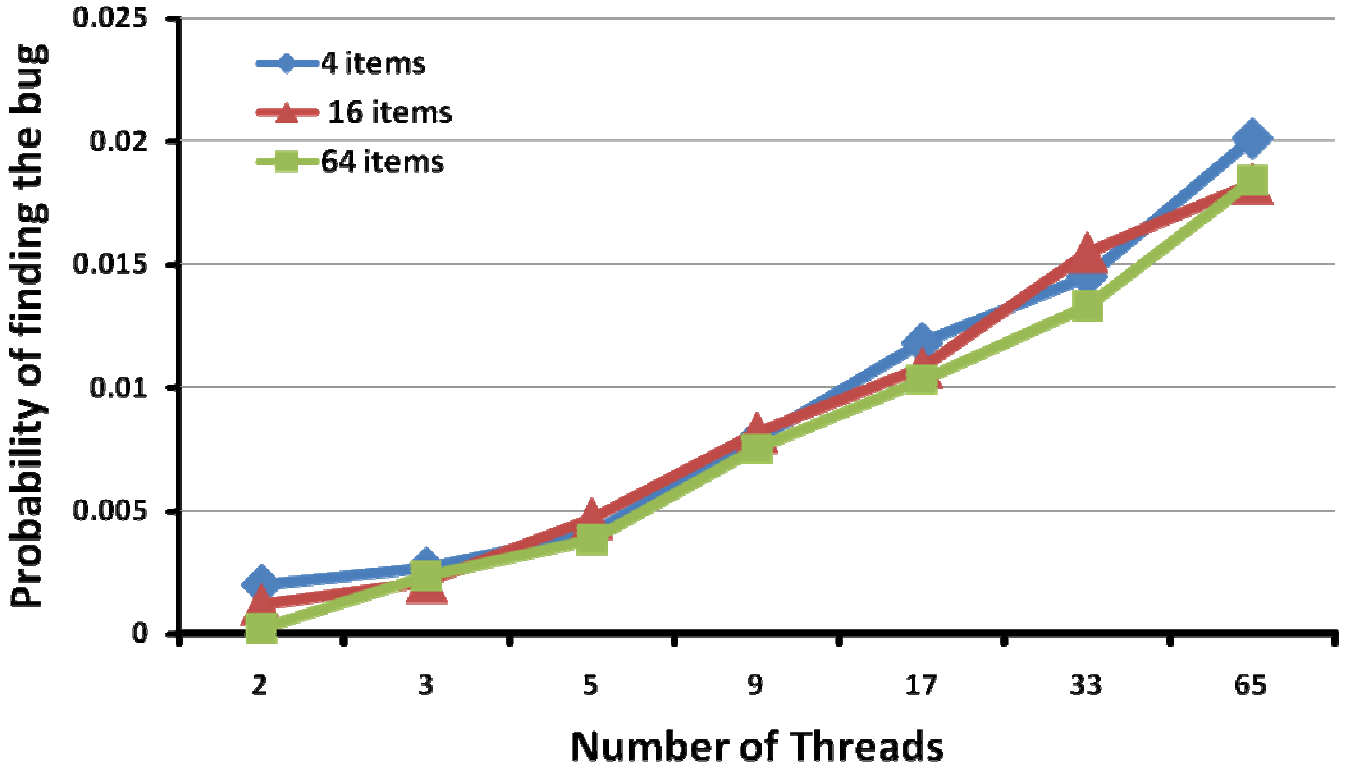
- Bug Depth = no. of ordering constraints sufficient to find the bug



- Probability of bug >= 1/nk
  - n: no. of threads (~ tens)
  - k: no. of instructions (~ millions)

Empirical bug probability w.r.t worst-case bound

- Probability increases with n, stays the same with k
  - In contrast, worst-case bound = $1/nk^{d-1}$



Why Cuzz is very effective

- Cuzz (probabilistically) finds all bugs in a single run
- Programs have *lots* of bugs
  - Cuzz is looking for all of them simultaneously
  - Probability of finding any of them is more than the probability of finding one
- Buggy code is executed many times
  - Each dynamic occurrence provides a new opportunity for Cuzz

Cuzz Status

- We are “dog fooding” Cuzz internally at Microsoft
- A non-commercial version will soon be available at
  - http://research.microsoft.com/projects/cuzz

CHESS: Systematic Enumeration of Thread Schedules

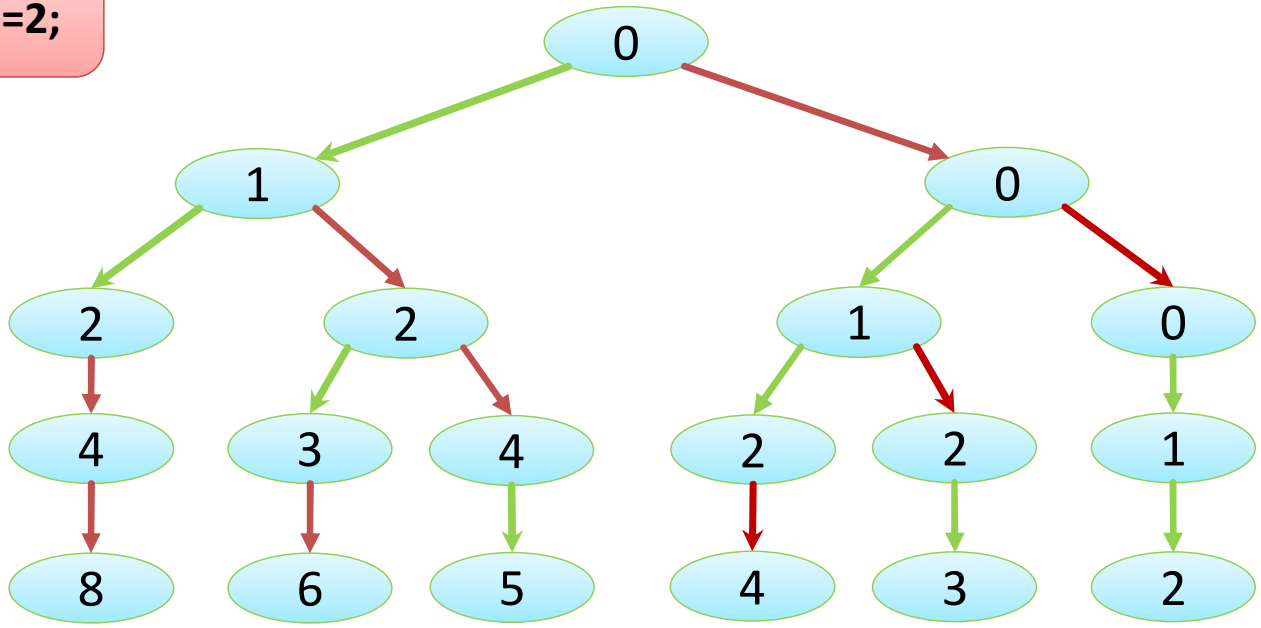
Refresh: Thread Schedule Space

Thread 1

```
x++;
x++;
```

Thread 2

```
x*=2;
x*=2;
```



Systematic Exploration

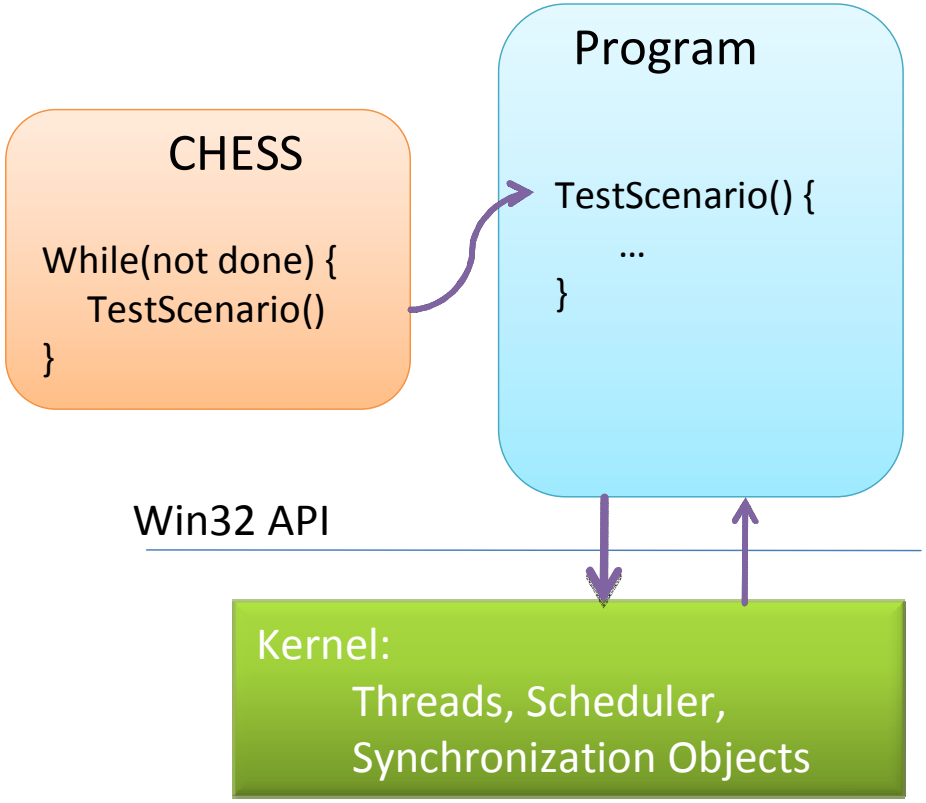
- If your program is small enough
- We can systematically enumerate thread schedules
  - Using model checking techniques

- Systematic enumeration is more efficient (in the long run) than random exploration

“Unit Testing” for Concurrency

- Idenitfy individual concurrency scenarios
  - e.g. spell-checking thread races with the rendering thread
- Test each scenario at a time
  - A test that creates the spell-checker and the renderer
  - Write assertions that look for correct behavior
- Contrast with stress/concurrency testing
  - Run multiple scenarios at the same time

Concurrency Unit Testing with CHESS

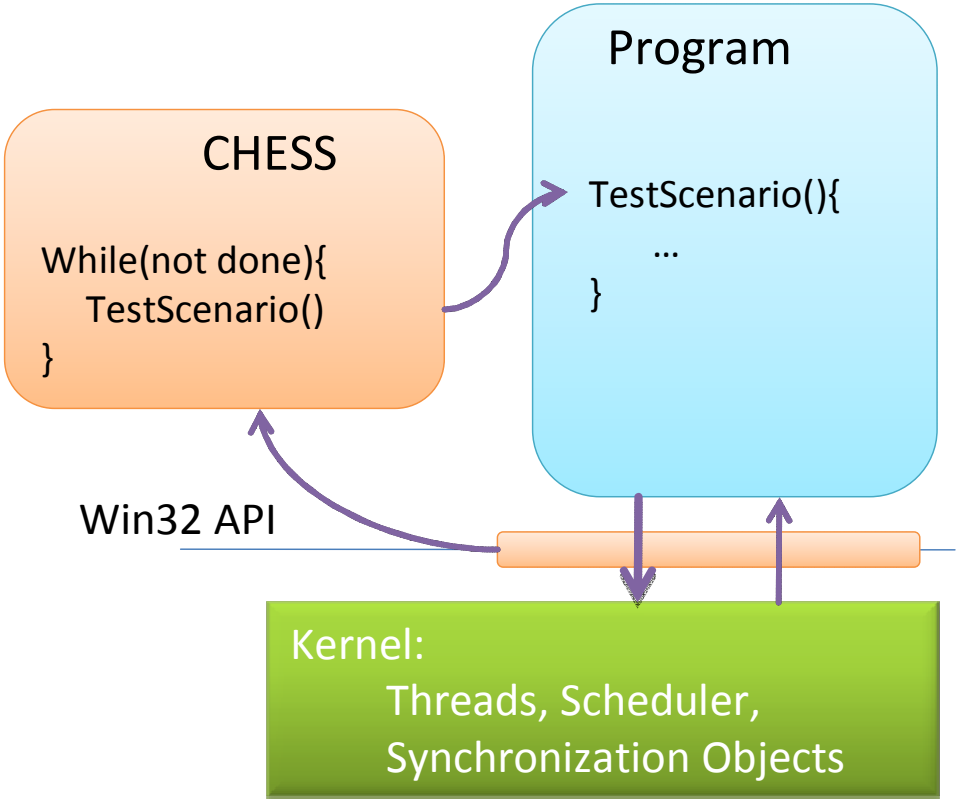


Tester Provides a Test Scenario

CHESS runs the scenario in a loop

- Every run takes a different interleaving
- Every run is repeatable

CHESS architecture



Run the system as is

- On the actual OS, hardware
- Using system threads, synchronization

Detour Win32 API calls

- To control and introduce nondeterminism
- CHESS 'hijacks' the scheduler

Advantages

- Avoid reporting false errors
- Easy to add to existing test frameworks
- Use existing debuggers

Dealing with State Space Explosion

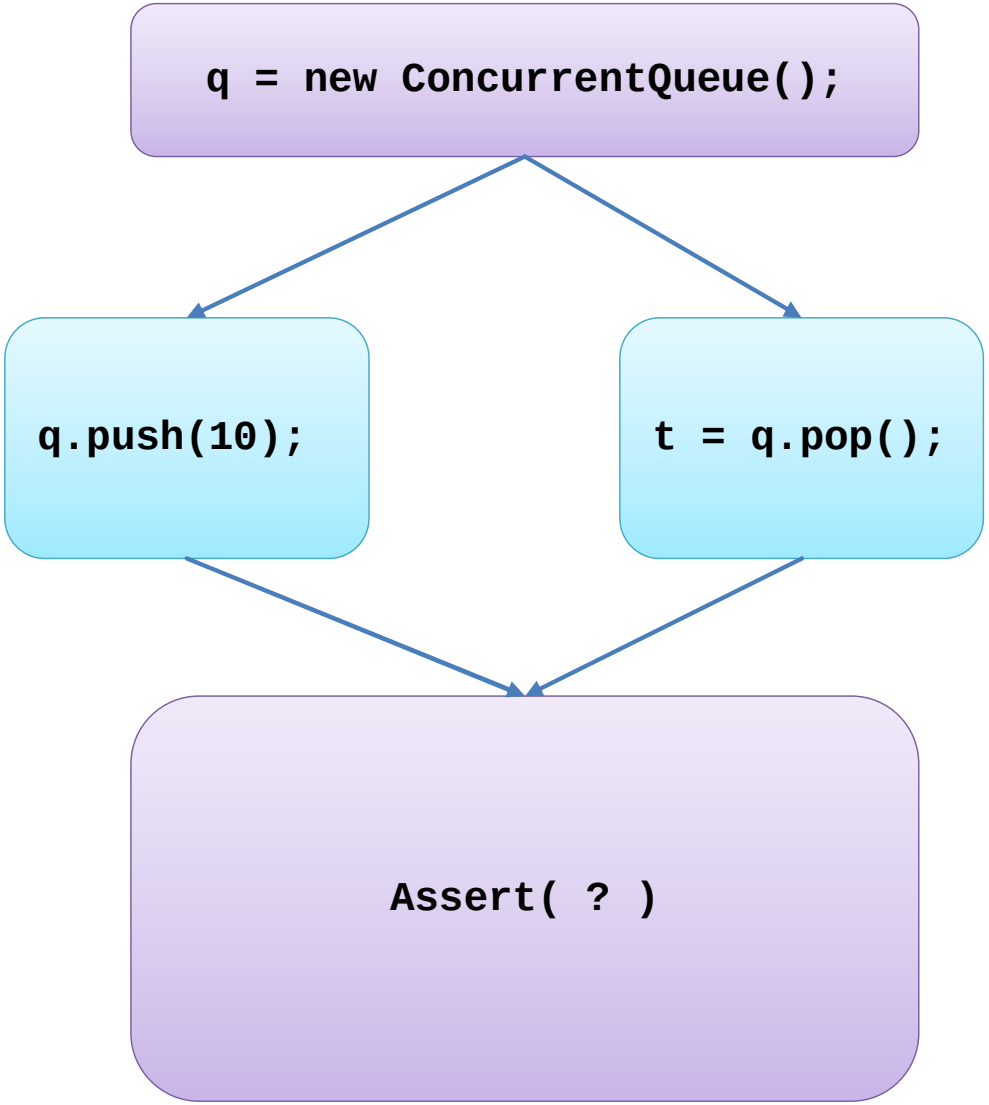
- CHESS implements many techniques to effectively explore the (astronomically) large state space
- Reduction techniques
  - Identify (exponentially many) schedules that have the same behavior
- Prioritization techniques
  - Derandomized version of Cuzz

CHESS availability

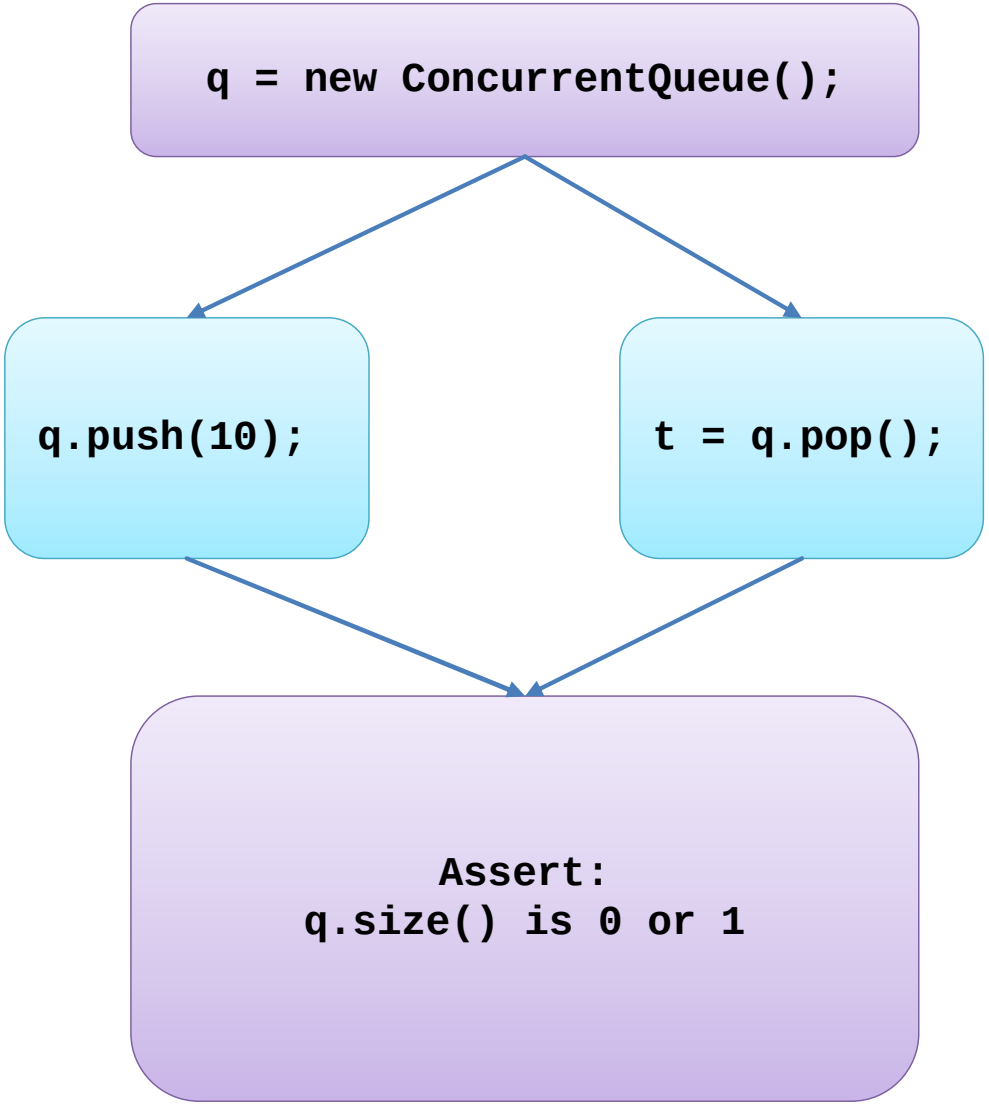
- Source available at
  - http://chesstool.codeplex.com

Automatic Thread Safety Checking

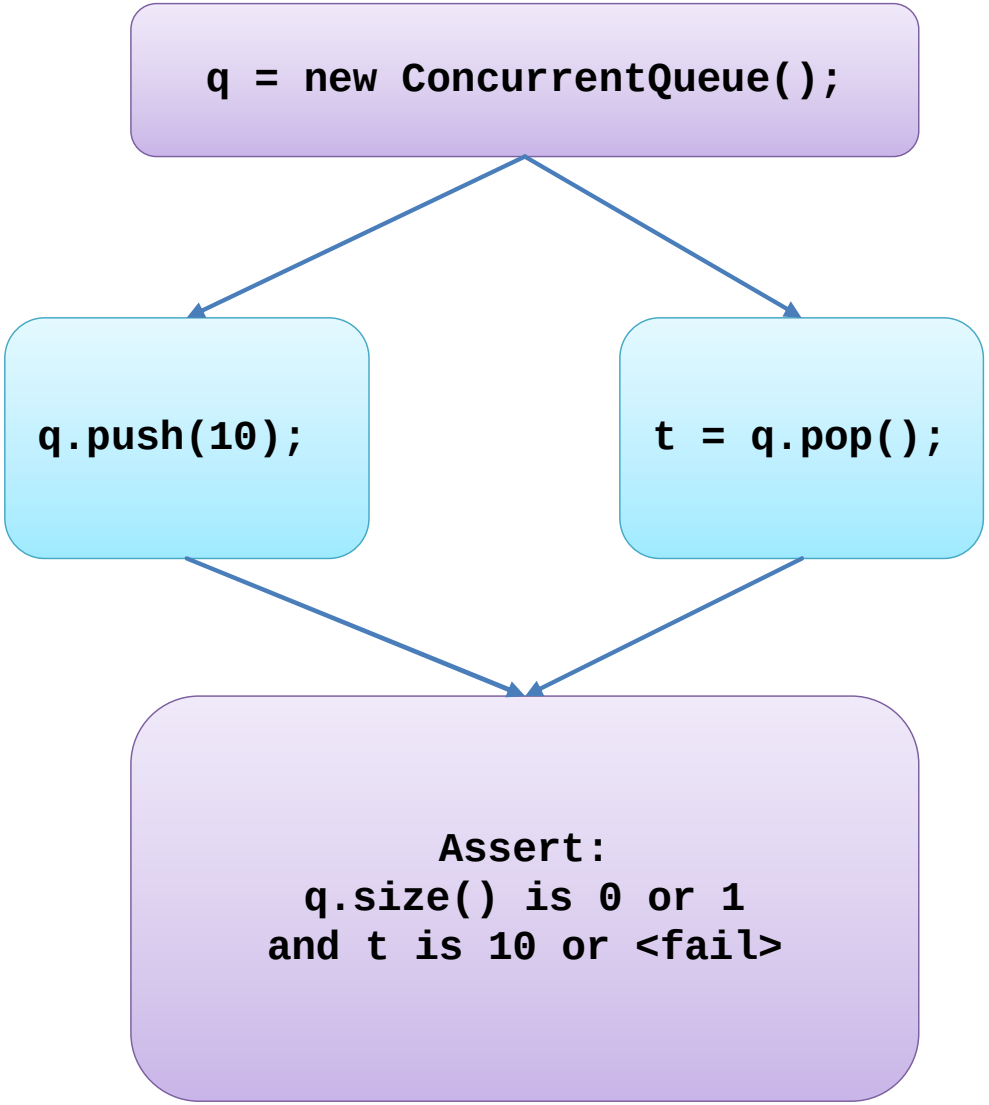
Let’s Write a Test



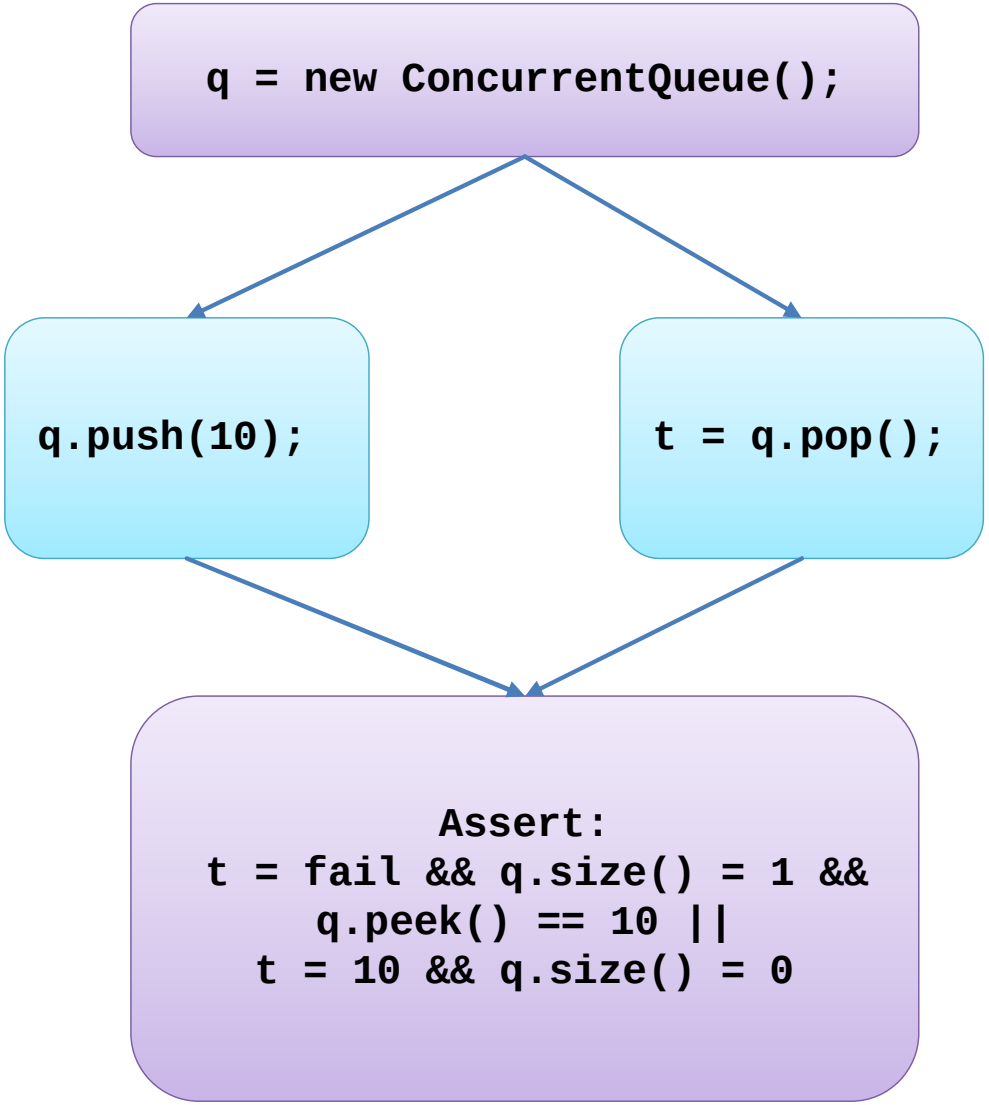
Let’s Write a Test



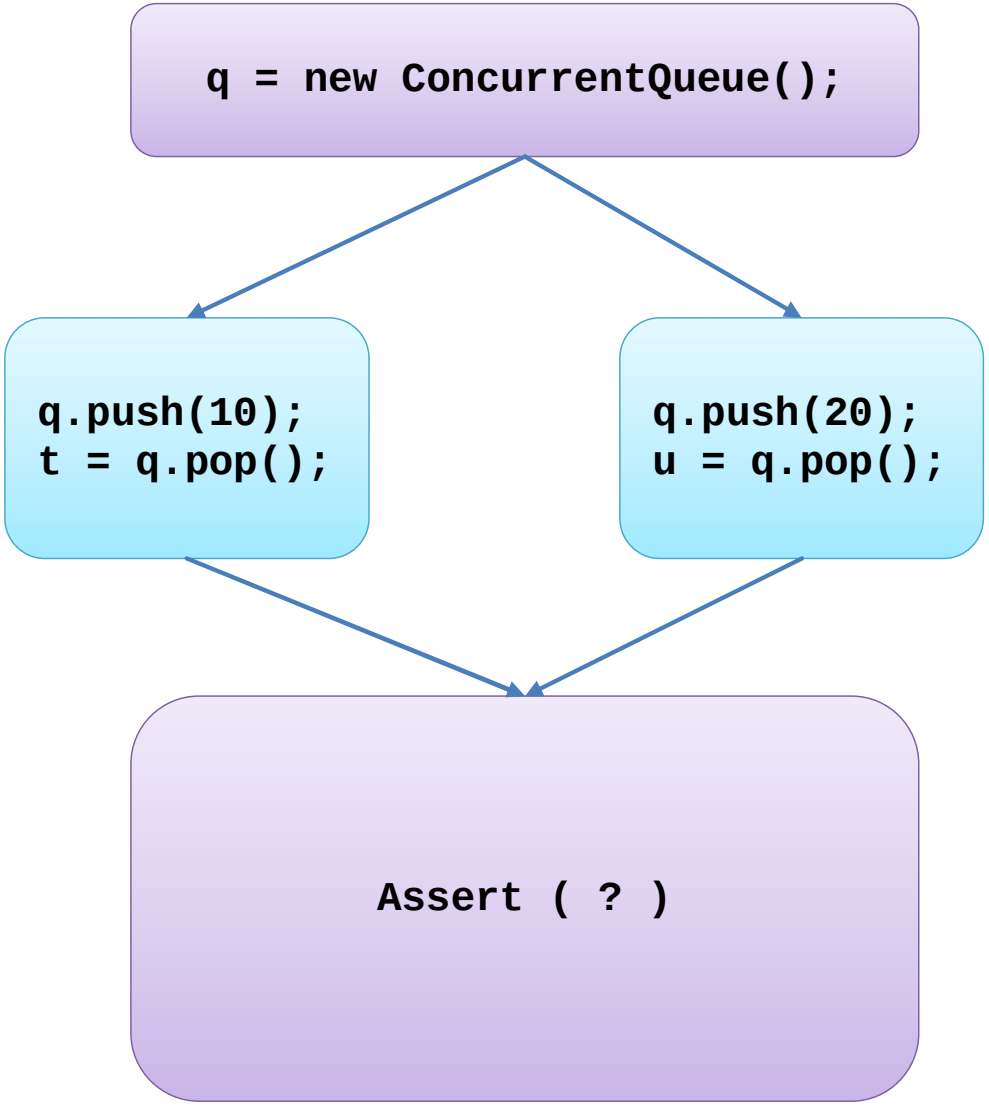
Let's Write a Test



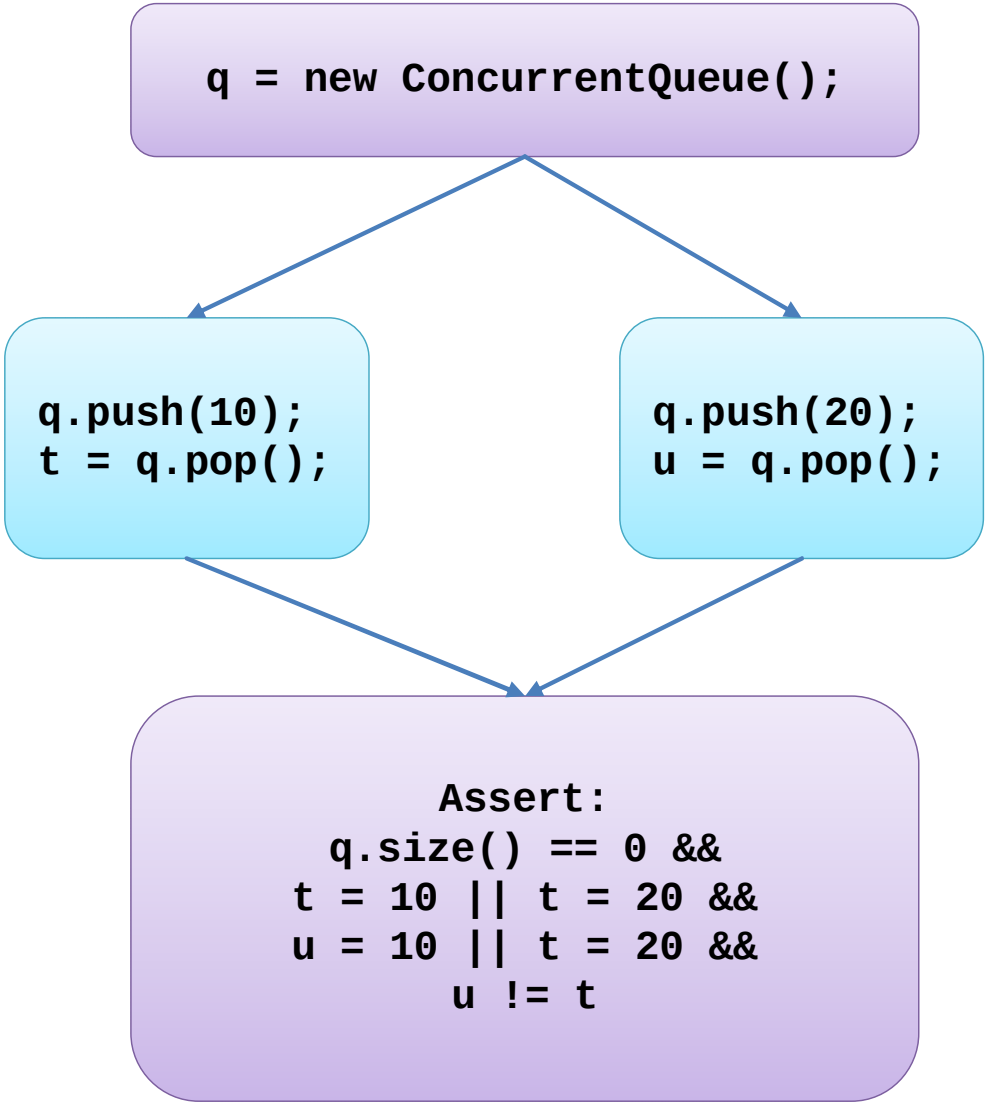
Let's Write a Test



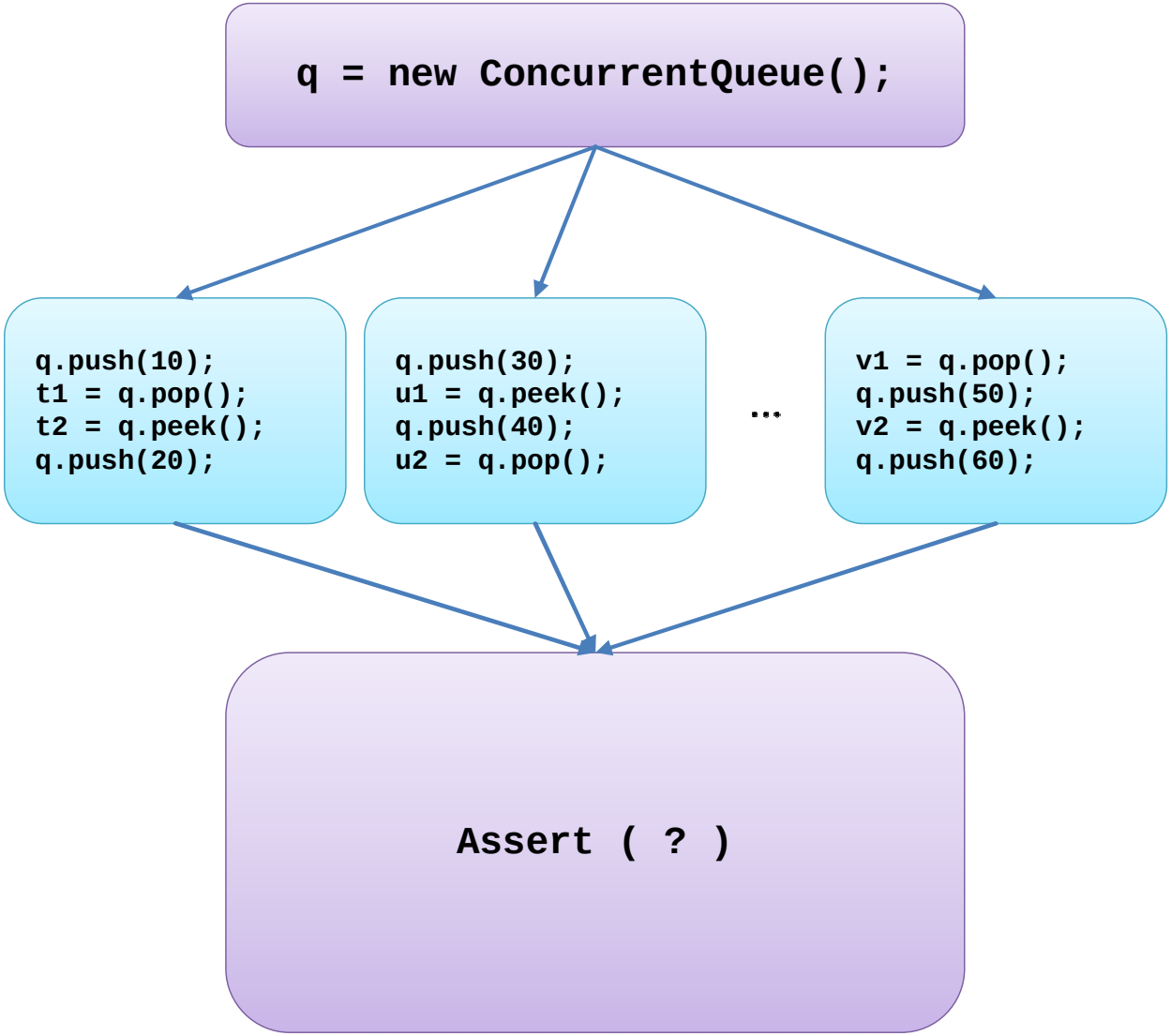
Let’s Write a Test



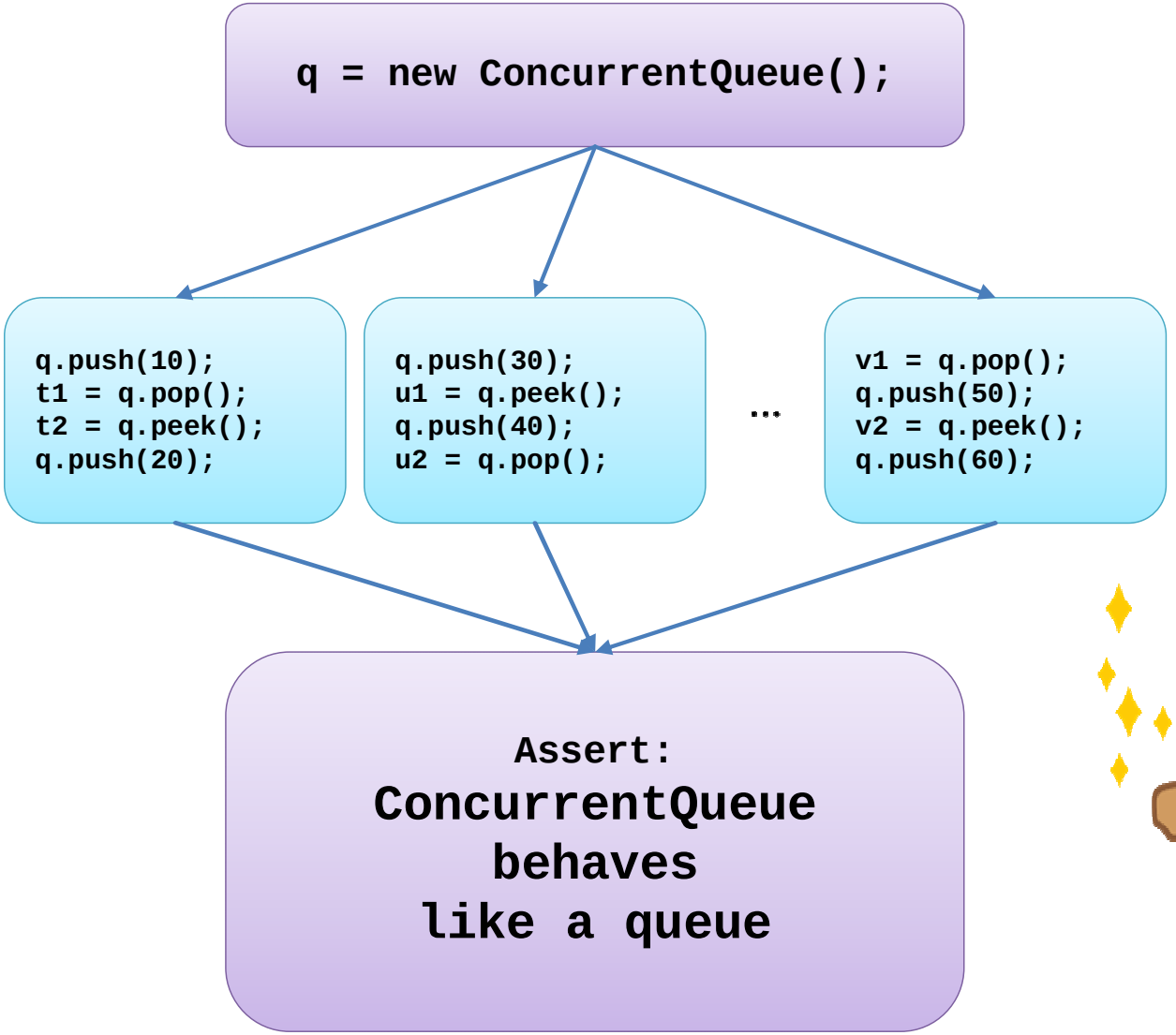
Let’s Write a Test



Let’s Write a Test



Wouldn’t it be nice if we could just say…

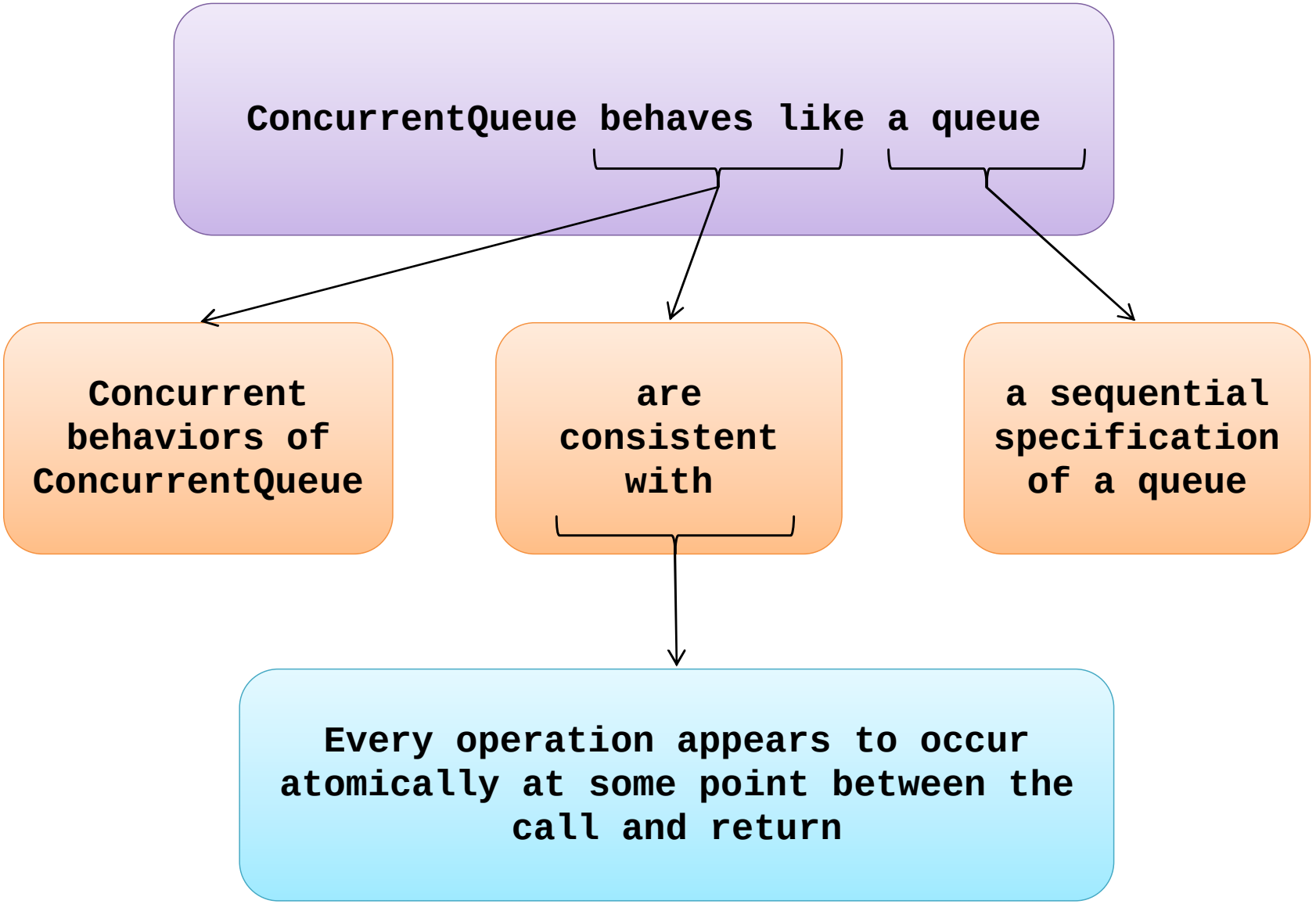


Informally, this is “thread safety”

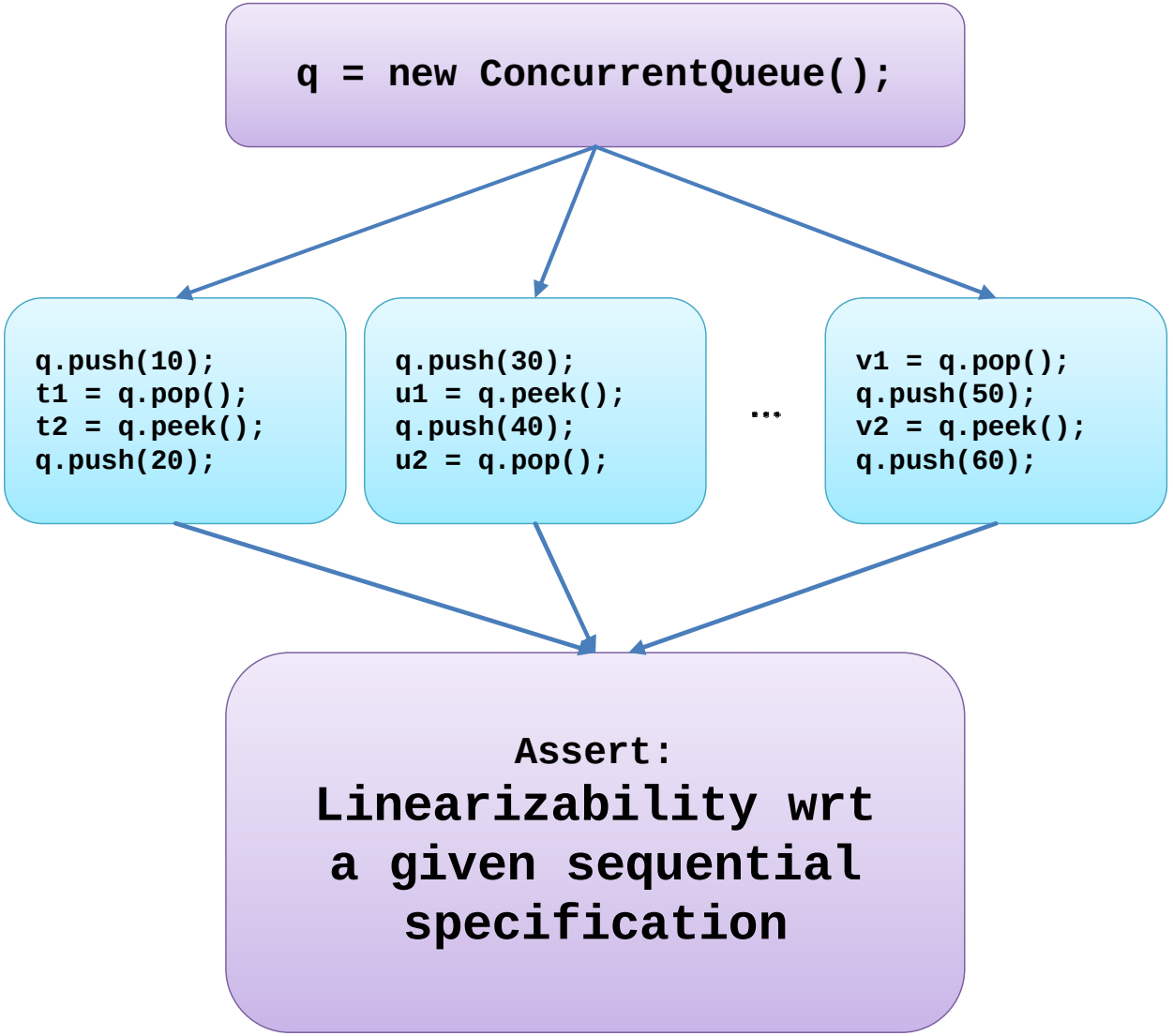
`ConcurrentQueue behaves like a queue`

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

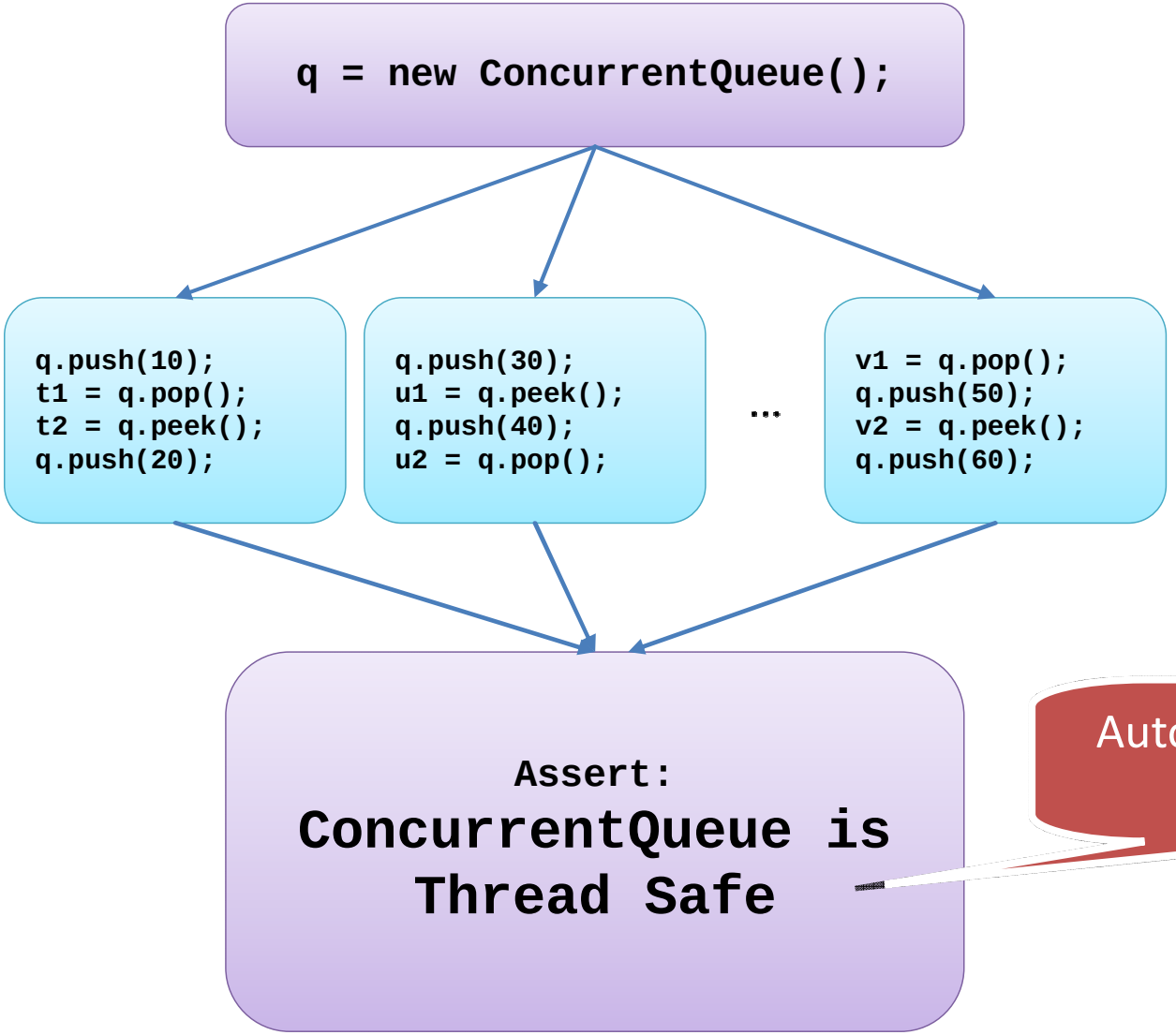
Formally, this is “Linearizability” [Herlihy & Wing ‘90]



So, simply check linearizability



Automatic Thread Safety Checking



Automatically learn how “a queue” behaves

Conclusions

- Beware of race conditions when you are designing your programs
  - Think of all source of nondeterminism
  - Reason about the space of program behaviors
- Use tools to explore the space
  - Cuzz: Randomized algorithm for large programs
  - CHESS: systematic algorithm for unit testing
  - Thread-safety as a correctness criterion