PURDUE
UNIVERSITY

Principles of Concurrency

Lecture 20
Testing

slides adapted from
https://www.research.ibm.com/haifa/Workshops/padtad2008/present/musuvathi.ppt

Background

Finding and Reproducing Heisenbugs in Concurrent Programs

Madanlal Musuvathi, Microsoft Research
Shaz Qadeer, Microsoft Research
Thomas Ball, Microsoft Research
Gerard Basler, ETH Zurich
Piramanayagam Arumuga Nainar, University of Wisconsin, Madison
Iulian Neamtiu, University of California, Riverside

Abstract

Concurrency is pervasive in large systems. Unexpected interference among threads often results in "Heisenbugs" that are extremely difficult to reproduce and eliminate. We have implemented a tool called CHESS for finding and reproducing such bugs. When attached to a program, CHESS takes control of thread scheduling and uses efficient search techniques to drive the program through possible thread interleavings. This systematic exploration of program behavior enables CHESS to quickly uncover bugs that might otherwise have remained hidden for a long time. For each bug, CHESS consistently reproduces an erroneous execution manifesting the bug, thereby making it significantly easier to debug the problem. CHESS scales to large concurrent programs and has found numerous bugs in existing systems that had been tested extensively prior to being tested by CHESS. CHESS has been integrated into the test frameworks of many code bases inside Microsoft and is used by testers on a daily basis.

1 Introduction

Building concurrent systems is hard. Subtle interactions among threads and the timing of asynchronous events can result in concurrency errors that are hard to find, reproduce, and debug. Stories are legend of so-called "Heisenbugs" [18] that occasionally surface in systems that have otherwise been running reliably for months. Slight changes to a program, such as the addition of debugging statements, sometimes drastically reduce the likelihood of erroneous interleavings, adding frustration to the debugging process.

The main contribution of this paper is a new tool called CHESS for systematic and deterministic testing of concurrent programs. When attached to a concurrent program, CHESS takes complete control over the scheduling of threads and asynchronous events, thereby capturing *all* the interleaving nondeterminism in the program. This provides two important benefits. First, if an execution results in an error, CHESS has the capability to reproduce the erroneous thread interleaving. This substantially improves the debugging experience. Second, CHESS uses systematic enumeration techniques [10, 37, 17, 31, 45, 22] to force every run of the program along a different thread interleaving. Such a systematic exploration greatly increases the chances of finding errors in existing tests. ...

Iterative Context Bounding for Systematic Testing of Multithreaded Programs

Madan Musuvathi, Shaz Qadeer
Microsoft Research
{madanm,qadeer}@microsoft.com

Keywords Concurrency, context-bounding, model checking, multithreading, partial-order reduction, shared-memory programs, software testing

1. Introduction

Multithreaded programs are difficult to get right. Specific thread interleavings, unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is *stress-testing*, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. ...

A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs

Sebastian Burckhardt, Microsoft Research, sburckha@microsoft.com
Pravesh Kothari, Indian Institute of Technology, Kanpur, praveshk@iitk.ac.in
Madanlal Musuvathi, Microsoft Research, madanm@microsoft.com
Santosh Nagarakatte, University of Pennsylvania, santoshn@cis.upenn.edu

... current program and an input test harness, PCT randomly schedules the threads of the program during each test run. In contrast to prior randomized testing techniques, PCT uses randomization sparingly and in a disciplined manner. As a result, PCT provides efficient mathematical probability of finding a concurrency bug in each run. Repeated independent runs can increase the probability of finding bugs to any user-desired level of certainty. In this paper, we demonstrate the ability of PCT to find bugs both theoretically, by stating and proving the probabilistic guarantees, and empirically, by applying PCT to several production-scale concurrent programs.

At the outset, it may seem impossible to provide effective probabilistic guarantees without exercising an astronomical number of schedules. Let us take a program with n threads that together execute at most k instructions. This program, to the first-order of approximation, has n^k possible thread schedules. If an adversary picks any one of these schedules to be the only buggy schedule, then no randomized scheduler can find the bug with a probability greater than $1/n^k$. Given that realistic programs create tens of threads (n) and can execute millions, if not billions, of instructions (k), such a bound is not useful.

PCT relies on the crucial observation that bugs in practice are not adversarial. Concurrency bugs typically involve unexpected interactions among few instructions executed by a small number of threads [19, 20]. If PCT is able to schedule these few instructions correctly, it succeeds in finding the bug irrespective of the numerous ways it can schedule instructions irrelevant to the bug. The following characterization of concurrency bugs captures this intuition precisely.

We define the *depth* of a concurrency bug as the minimum number of scheduling constraints that are sufficient to find the bug. Intuitively, bugs with a higher depth exhibit in fewer schedules and are thus inherently harder to find. Fig. 1 explains this through a series of examples. The bug in Fig. 1(a) manifests whenever Thread 2 accesses t before the initialization by Thread 1. We graphically represent this *ordering constraint* as an arrow. Any schedule that satisfies this ordering constraint finds the bug irrespective of the ordering of other instructions in the program. By our definition, this bug is of depth 1. Fig. 1 shows two more examples of common concurrency errors, an atomicity violation in (b) and a deadlock in (c). Both these errors require two ordering constraints and are thus of depth 2.

The Heisenbug problem

Concurrent executions are highly nondeterminisitic

Rare thread interleavings result in Heisenbugs

- Difficult to find, reproduce, and debug

Observing the bug can "fix" it

- Likelihood of interleavings changes, say, when you add printfs

A huge productivity problem

- Developers and testers can spend weeks chasing a single Heisenbug

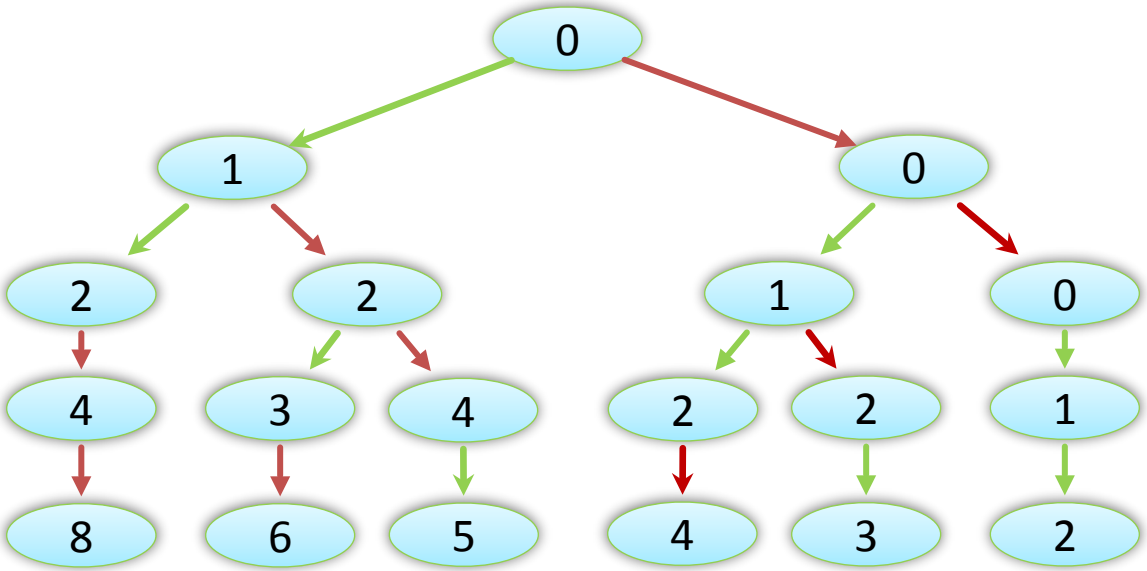
Thread interleavings

Thread 1

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

Thread 2

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



CHESS in a nutshell

CHESS is a user-mode scheduler

Controls all scheduling nondeterminism

- Replace the OS scheduler

Guarantees:

- Every program run takes a different thread interleaving
- Reproduce the interleaving for every run

High level goals

Scale to large programs

Any error found by CHESS is possible in the wild

- CHESS does not introduce any new behaviors

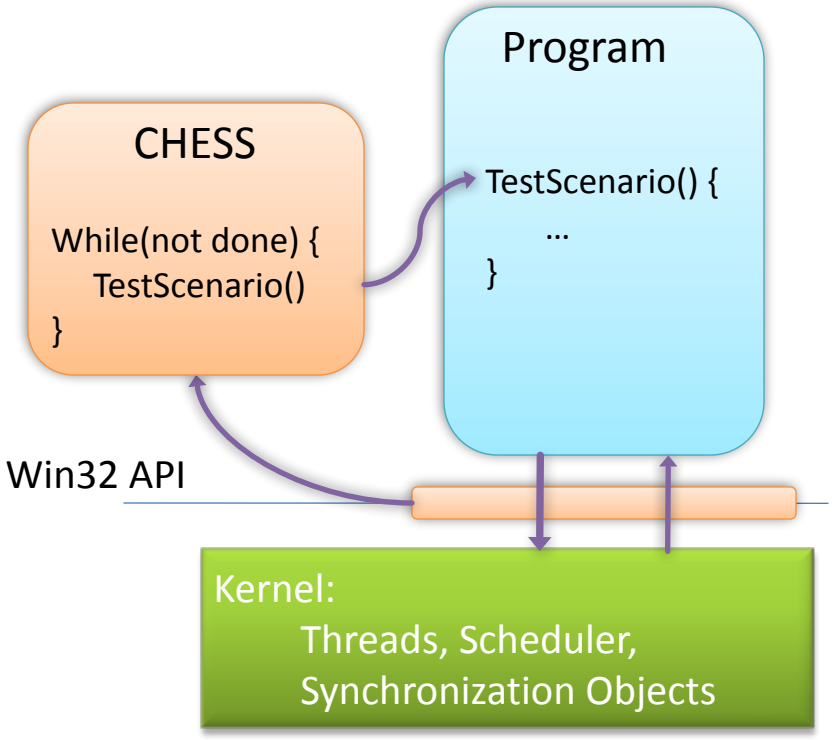
Any error found in the wild can be found by CHESS

- Need to capture all sources of nondeterminism
- Exhaustively explore the nondeterminism (state explosion)

  e.g. Enumerate all thread interleavings
- Hard to achieve

  Practical goal: beat stress

CHESS architecture



CHESS runs the scenario in a loop

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

Intercept synch. & threading calls

- To control and introduce nondeterminism

Detect

- Assertion violations
- Deadlocks
- Dataraces
- Livelocks

Running Example

Thread 1

```
Lock (l);
bal += x;
Unlock(l);
```

Thread 2

```
Lock (l);
t = bal;
Unlock(l);

Lock (l);
bal = t - y;
Unlock(l);
```

Introduce Schedule() points

Thread 1

```
Schedule();
Lock (l);
bal += x;
Schedule();
Unlock(l);
```

Thread 2

```
Schedule(); Lock
(l);
t = bal;
Schedule();
Unlock(l);

Schedule(); Lock
(l);
bal = t - y;
Schedule();
Unlock(l);
```

- Instrument calls to the CHESS scheduler
- Each call is a potential preemption point

First-cut solution: Random sleeps

Introduce random sleep at schedule points

Does not introduce new behaviors

- Sleep models a possible preemption at each location
- Sleeping for a finite amount guarantees starvation-freedom

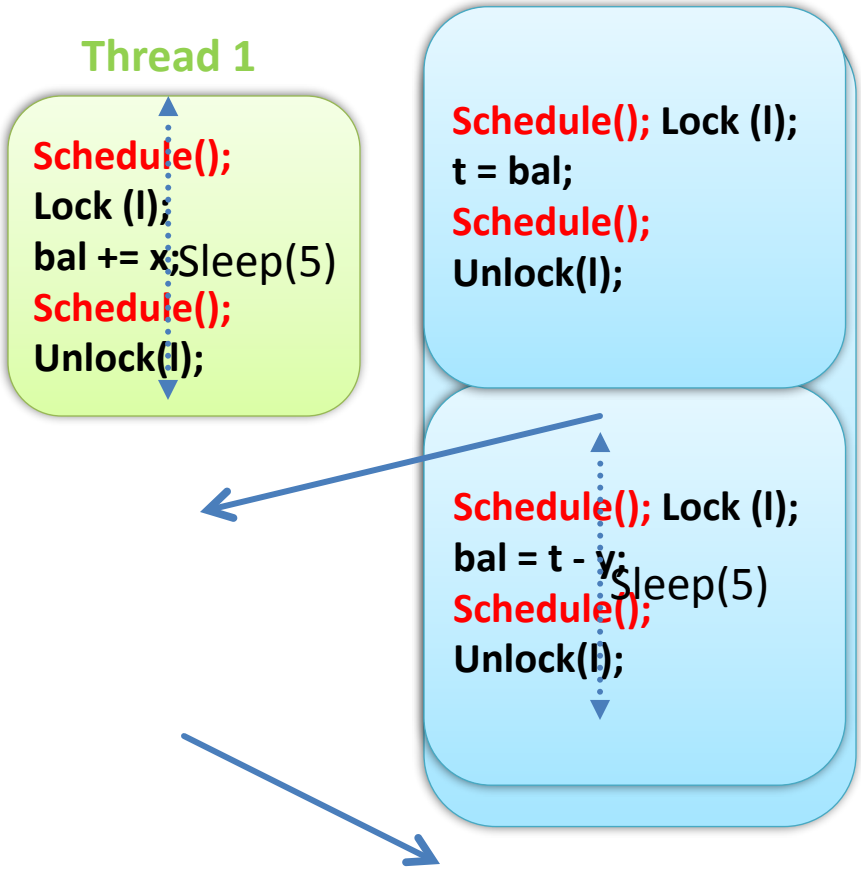
Thread 1

```
Sleep(rand());
Lock (l);
bal += x;
Sleep(rand());
Unlock(l);
```

Thread 2

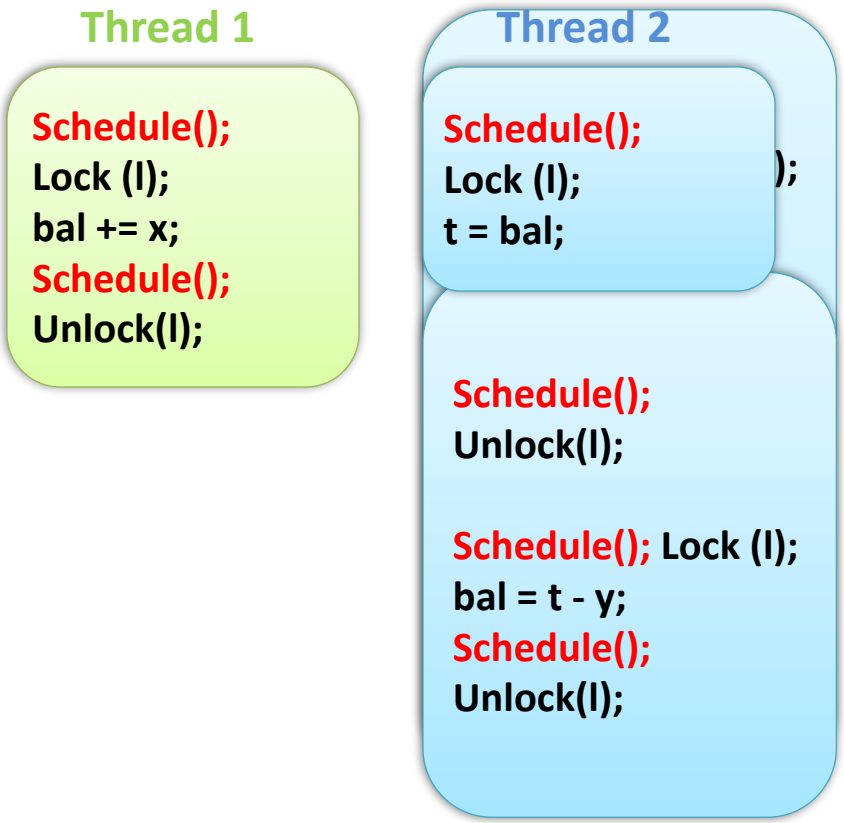
```
Sleep(rand());
Lock (l);
t = bal;
Sleep(rand());
Unlock(l);

Sleep(rand());
Lock (l);
bal = t - y;
Sleep(rand());
Unlock(l);
```

Improvement 1:



- Delays that result in the same "happens-before" graph are equivalent
- Avoid exploring equivalent interleavings

Improvement 2:



- Avoid exploring delays that are impossible
- Identify when threads can make progress
- CHESS maintains a run queue and a wait queue
  - Mimics OS scheduler state

Emulate execution on a uniprocessor

Thread 1

```
Schedule();
Lock (l);
bal += x;
Schedule();
Unlock(l);
```

Thread 2

```
Schedule(); Lock
(l);
t = bal;
Schedule();
Unlock(l);
```

```
Schedule(); Lock
(l);
bal = t - y;
Schedule();
Unlock(l);
```

- Enable only one thread at a time
- Linearizes a partial-order into a total-order
- Controls the order of data-races

Capture all sources of nondeterminism?

Scheduling nondeterminism? Yes

Timing nondeterminism? Yes

- Controls when and in what order the timers fire

Nondeterministic system calls? Mostly

- CHESS uses precise abstractions for many system calls

Input nondeterminism? No

- Rely on users to provide inputs

Program inputs, return values of system calls, files read, packets received,…

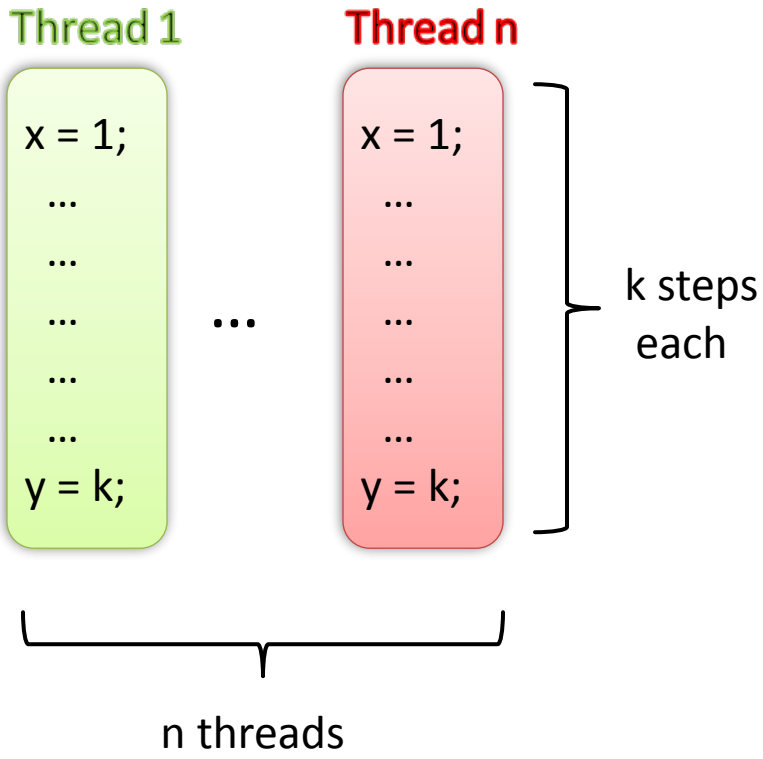
- Good tradeoff in the short term

But can't find race-conditions on error handling code

- Future extensions using symbolic execution?

(DART, jCUTE, SAGE, PEX)

State space explosion



Number of executions

$= O(n^{nk})$

Exponential in both n and k

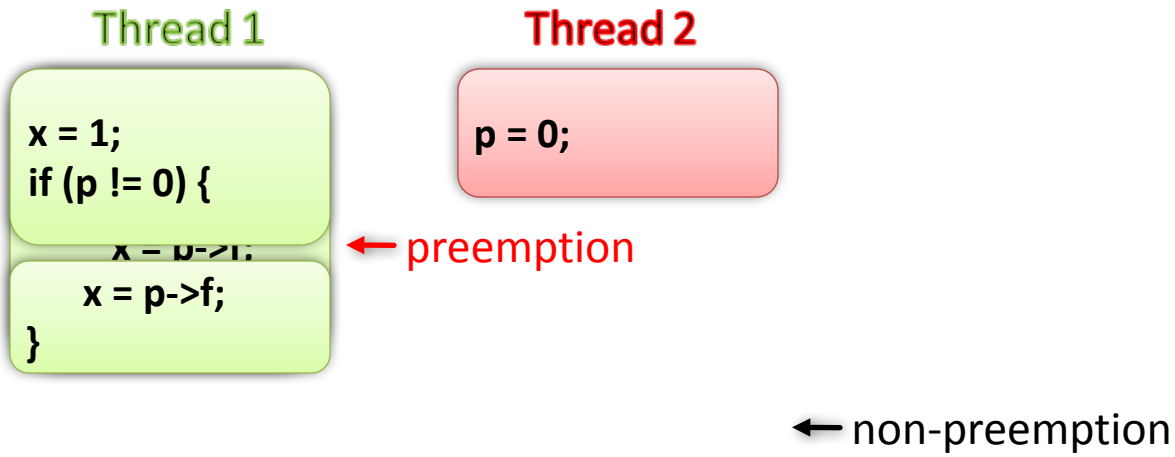
- Typically: $n < 10$ $k > 100$

Limits scalability to large programs

Goal: Scale CHESS to large programs (large k)

Preemption bounding

- Prioritize executions with small number of preemptions
- Two kinds of context switches:
  - Preemptions – forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end



So, is CHESS is unsound?

Soundness: prove that the program is correct for a given input test harness

- Need to exhaustively explore all interleavings

For small programs, CHESS is sound

- Iteratively increase the preemption bound

Preemption bounding helps scale to large programs

- A good "knob" to trade resources for coverage

Better search algorithms → more coverage faster

- Partial-order reduction
- Modular testing of loosely-coupled programs

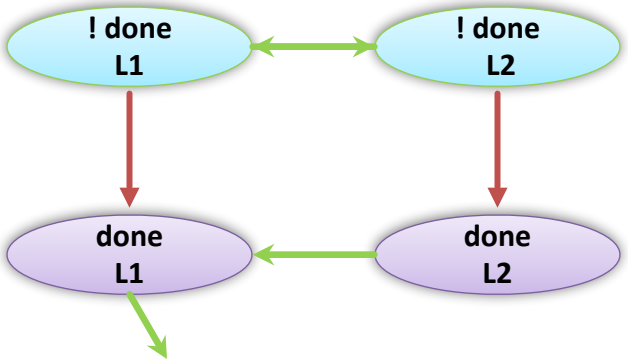
Concurrent programs have cyclic state spaces

Thread 1

```
L1:  while( ! done) {
L2:     Sleep();
       }
```

Thread 2

```
M1: done = 1;
```



- Spinlocks
- Non-blocking algorithms
- Implementations of synchronization primitives
- Periodic timers
- …

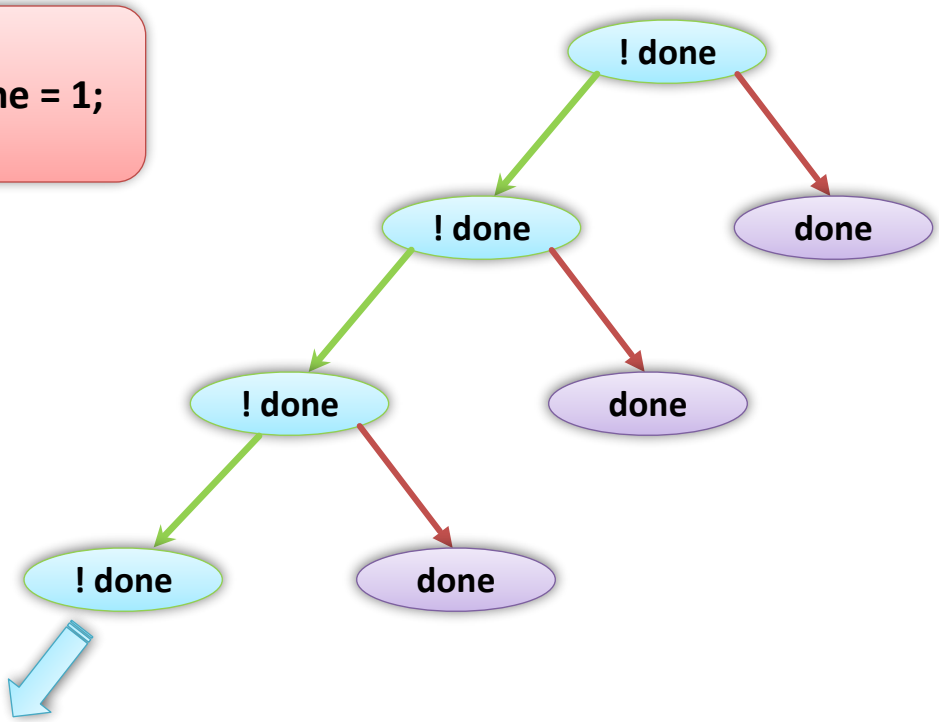
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

```
while( ! done)
{
    Sleep();
}
```

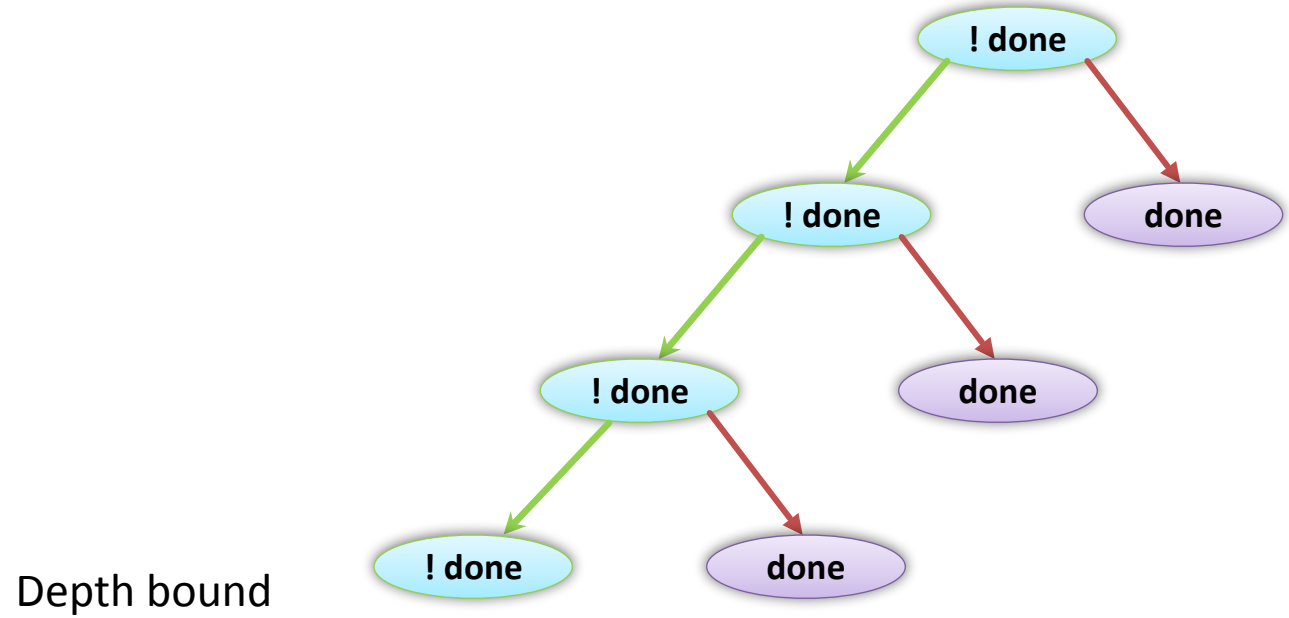
Thread 2

```
done = 1;
```



Depth bounding

Prune executions beyond a bounded number of steps

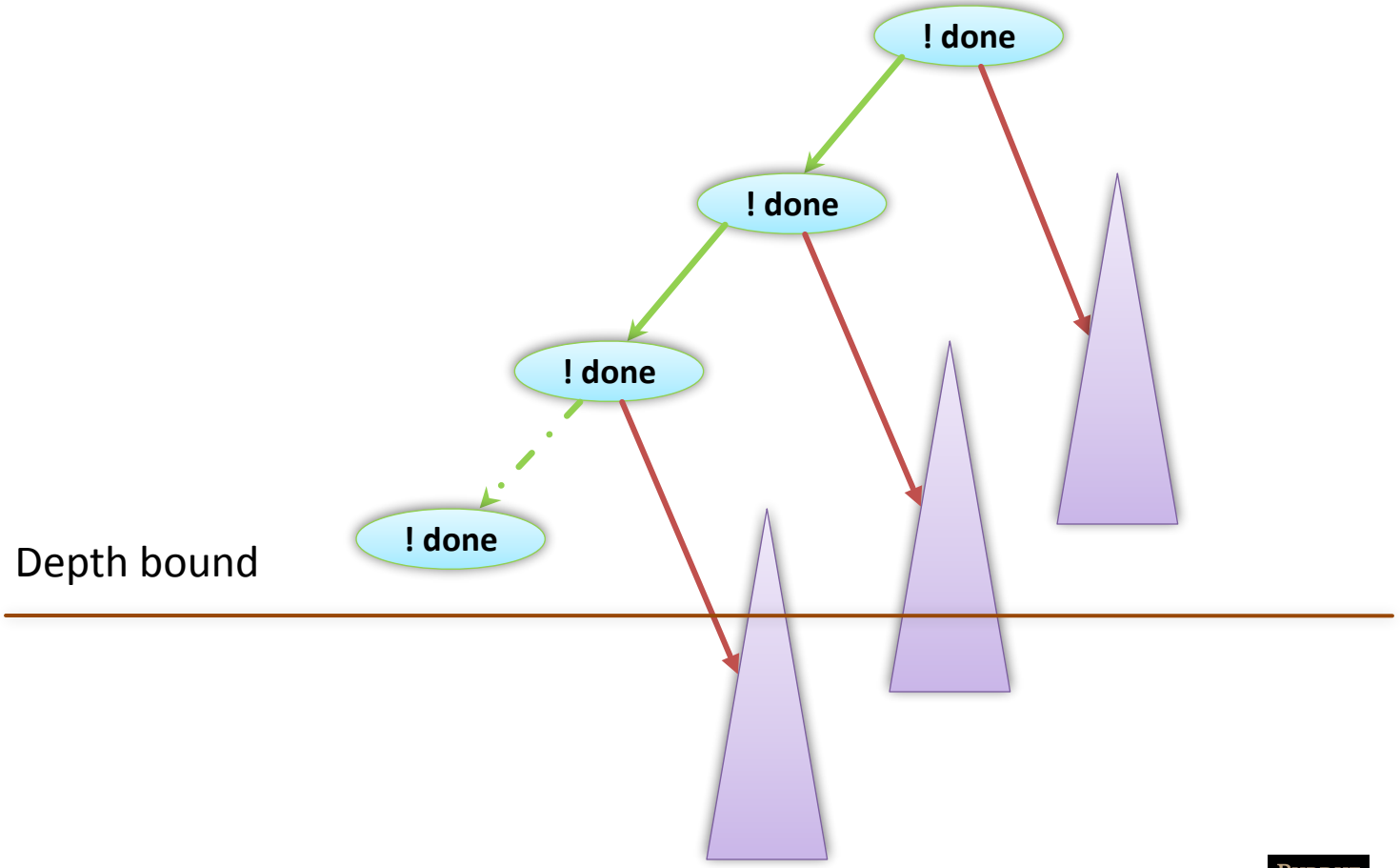


Problem 1: Ineffective state coverage

Bound has to be large enough to reach the deepest bug

- Typically, greater than 100 synchronization operations

Every unrolling of a cycle redundantly explores reachable state space



Problem 2: Cannot find livelocks

Livelocks : lack of progress in a program

Thread 1

```
temp = done;
while( ! temp)
{
    Sleep();
}
```

Thread 2

```
done = 1;
```

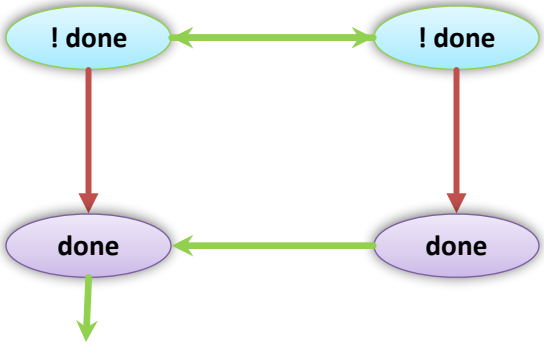
Key idea

Thread 1

```
while( ! done)
{
    Sleep();
}
```

Thread 2

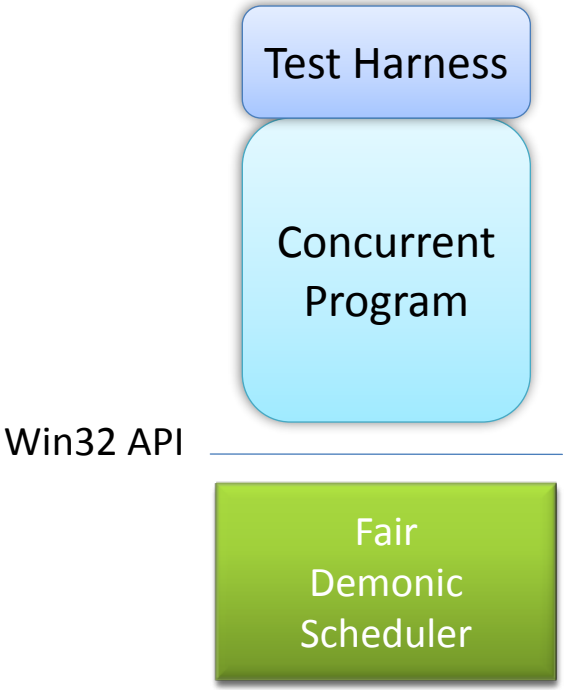
```
done = 1;
```



- This test terminates only when the scheduler is fair
- Fairness is assumed by programmers

All cycles in correct programs are unfair
A fair cycle is a livelock

We need a fair demonic scheduler



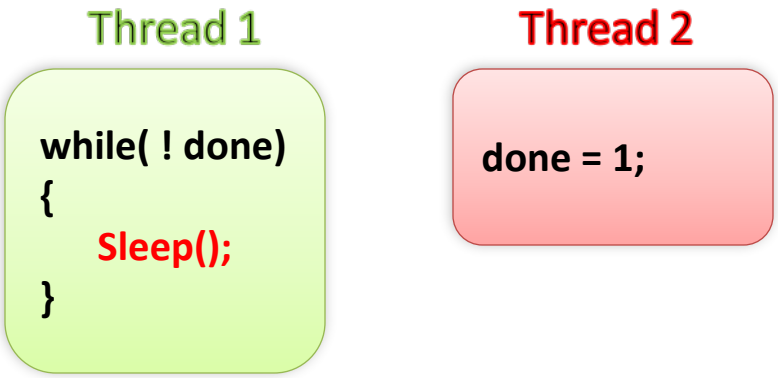
Avoid unrolling unfair cycles

- Effective state coverage

Detect fair cycles

- Find livelocks (violations of fair termination)

Fair termination allows CHESS to check for arbitrary liveness properties



Example: Good Samaritan assumption

- Forall threads t : scheduled(t) → yield(t)
- A thread when scheduled infinitely often yields the processor infinitely often

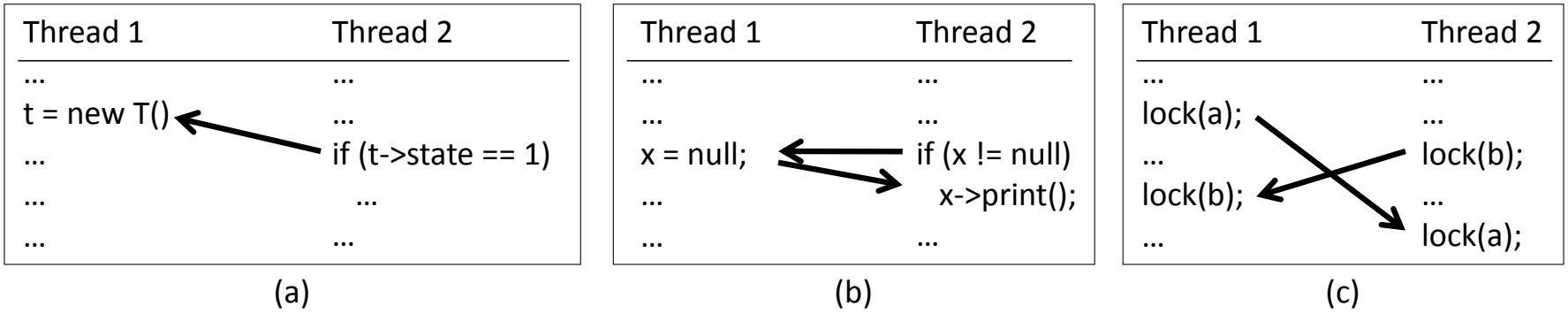
Examples of yield:

- Sleep(), ScheduleThread(), asm {rep nop;}
- Thread completion

Probabilistic Concurrency Testing

- Also a random scheduler, but uses randomization sparingly
  - Repeated independent runs increase probability of finding a bug
- Naive approach is exponential (n threads with k instructions has n^k possible thread schedules)
- But, bugs in practice are not adversarial
  - small number of instructions executed by small number of threads
  - goal: schedule these instructions correctly
- The depth of a concurrency bug is the minimum number of scheduling constraints sufficient to find the bug
  - PCT focusses on probabilistically finding bugs at a given depth
  - Can find a bug at depth d in $O(nk^{d-1})$ independent runs

Ordering Edges and Depth



- (a) manifests whenever the conditional in T2 executes before
- (b) manifests when two ordering constraints are satisfied
- (c) also manifests under two ordering constraints

Bug Depth

Main Thread	Filewriter Thread
…	…
free(mutex)	mutex.unlock()
exit(0);	

Depth 2

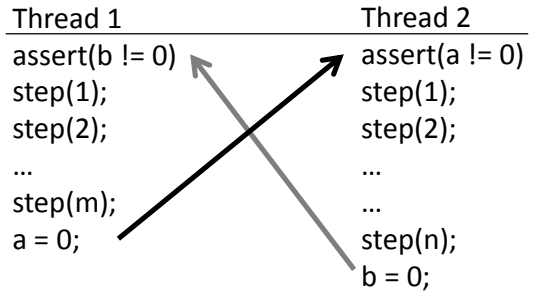
Thread 1	Thread 2
…	…
Set(e);	…
t = new T()	Wait(e);
…	t->state == 1
…	…

Depth 1

Thread 1	Thread 2
…	…
init = true	…
t = new T()	if (init)
…	t->state == 1
…	…

Depth 2

Randomization



Two bugs of depth 1 difficult to find using a pure randomized scheduler

The PCT Algorithm

- instead, use a priority-based scheduler
- the scheduler schedules a low priority thread only when all higher priority threads are blocked
- threads change priorities when they pass a priority change point
- randomly assign initial priority values and randomly pick priority change points