CHESS: Analysis and Testing of Concurrent Programs

Sebastian Burckhardt, Madan Musuvathi, Shaz Qadeer

Microsoft Research

Joint work with

Tom Ball, Peli de Halleux, and interns
Gerard Basler (ETH Zurich),
Katie Coons (U. T. Austin),
P. Arumuga Nainar (U. Wisc. Madison),
Iulian Neamtiu (U. Maryland, U.C. Riverside)

What you will learn in this tutorial

- Difficulties of testing/debugging multithreaded programs
- CHESS – verifier for multi-threaded programs
  - Provides systematic coverage of thread interleavings
  - Provides replay capability for easy debugging
- CHESS algorithms
- Types of concurrency errors, including data races
- How to extend CHESS
  - CHESS monitors

Concurrent Programming is HARD

- 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

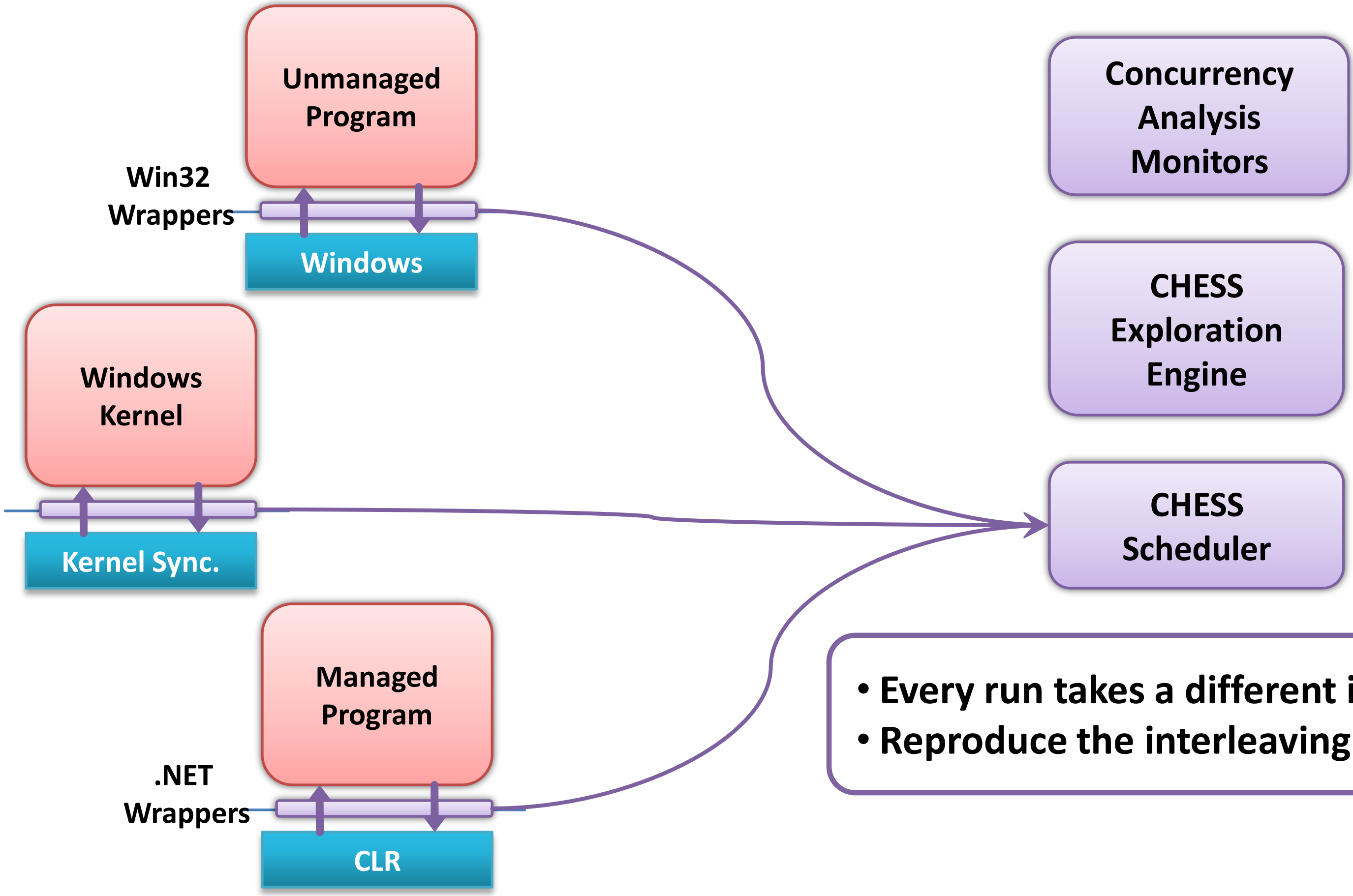
CHESS in a nutshell

- CHESS is a user-mode scheduler
  - Controls all scheduling nondeterminism
- Guarantees:
  - Every program run takes a different thread interleaving
  - Reproduce the interleaving for every run
- Provides monitors for analyzing each execution

CHESS Demo

- Find a simple Heisenbug

CHESS Architecture



Unmanaged Program

Win32 Wrappers

Windows

Windows Kernel

Kernel Sync.

Managed Program

.NET Wrappers

CLR

Concurrency Analysis Monitors

CHESS Exploration Engine

CHESS Scheduler

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

The Design Space for CHESS

- Scale
  - Apply to large programs
- Precision
  - Any error found by CHESS is possible in the wild
  - CHESS should not introduce any new behaviors
- Coverage
  - Any error found in the wild can be found by CHESS
  - Capture all sources of nondeterminism
  - Exhaustively explore the nondeterminism
- Generality of Specifications
  - Find interesting classes of concurrency errors
  - Safety and liveness

Comparison with other approaches to verification

	Model Checking	Static Analysis	CHESS
Scalability	+	++	++
Precision	+	+	++
Coverage	++	++	+
Generality	++	+	++

Errors that CHESS can find

- Assertions in the code
- Any dynamic monitor that you run
  - Memory leaks, double-free detector, …
- Deadlocks
  - Program enters a state where no thread is enabled
- Livelocks
  - Program runs for a long time without making progress
- Dataraces
- Memory model races

CHESS Scheduler

Concurrent Executions are Nondeterministic

Thread 1

```
x = 1;
y = 1;
```

Thread 2

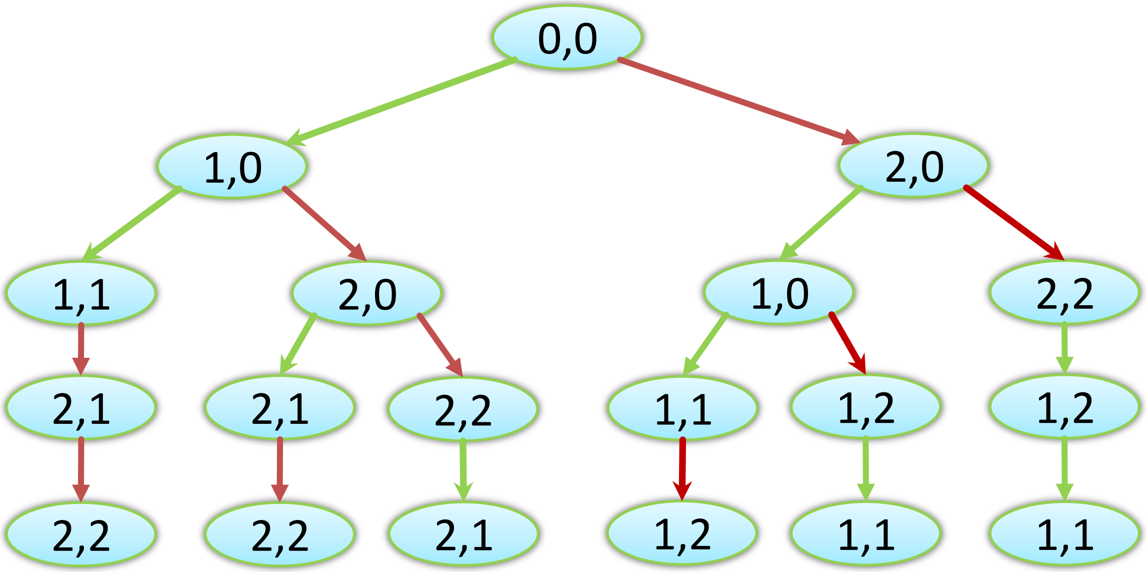
```
x = 2;
y = 2;
```

`x = 1;`

`y = 1;`

`x = 2;`

`y = 2;`



High level goals of the scheduler

- Enable CHESS on real-world applications
  - IE, Firefox, Office, Apache, …
- Capture all sources of nondeterminism
  - Required for reliably reproducing errors
- Ability to explore these nondeterministic choices
  - Required for finding errors

Sources of Nondeterminism

1. Scheduling Nondeterminism

- Interleaving nondeterminism
  - Threads can race to access shared variables or monitors
  - OS can preempt threads at arbitrary points

- Timing nondeterminism
  - Timers can fire in different orders
  - Sleeping threads wake up at an arbitrary time in the future
  - Asynchronous calls to the file system complete at an arbitrary time in the future

Sources of Nondeterminism

1. Scheduling Nondeterminism

- Interleaving nondeterminism
  - Threads can race to access shared variables or monitors
  - OS can preempt threads at arbitrary points

- Timing nondeterminism
  - Timers can fire in different orders
  - Sleeping threads wake up at an arbitrary time in the future
  - Asynchronous calls to the file system complete at an arbitrary time in the future
- CHESS captures and explores this nondeterminism

Sources of Nondeterminism
2. Input nondeterminism

- User Inputs
  - User can provide different inputs
  - The program can receive network packets with different contents

- Nondeterministic system calls
  - Calls to gettimeofday(), random()
  - ReadFile can either finish synchronously or asynchronously

Sources of Nondeterminism

2. Input nondeterminism

- User Inputs
  - User can provide different inputs
  - The program can receive network packets with different contents
  - CHESS relies on the user to provide a scenario
- Nondeterministic system calls
  - Calls to gettimeofday(), random()
  - ReadFile can either finish synchronously or asynchronously
  - CHESS provides wrappers for such system calls

Sources of Nondeterminism
3. Memory Model Effects

- Hardware relaxations
  - The processor can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program

- Compiler relaxations
  - Compiler can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program (with data races)

Sources of Nondeterminism

3. Memory Model Effects

- Hardware relaxations
  - The processor can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program
  - CHESS contains a monitor for detecting such relaxations
- Compiler relaxations
  - Compiler can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program (with data races)
  - Future Work

Interleaving Nondeterminism: Example

```
init:
  balance = 100;
```

Deposit Thread

```
void Deposit100(){
  EnterCriticalSection(&cs);
  balance += 100;
  LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```
void Withdraw100(){
  int t;

  EnterCriticalSection(&cs);
  t = balance;
  LeaveCriticalSection(&cs);

  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);

}
```

```
final:
  assert(balance = 100);
```

Invoke the Scheduler at Preemption Points

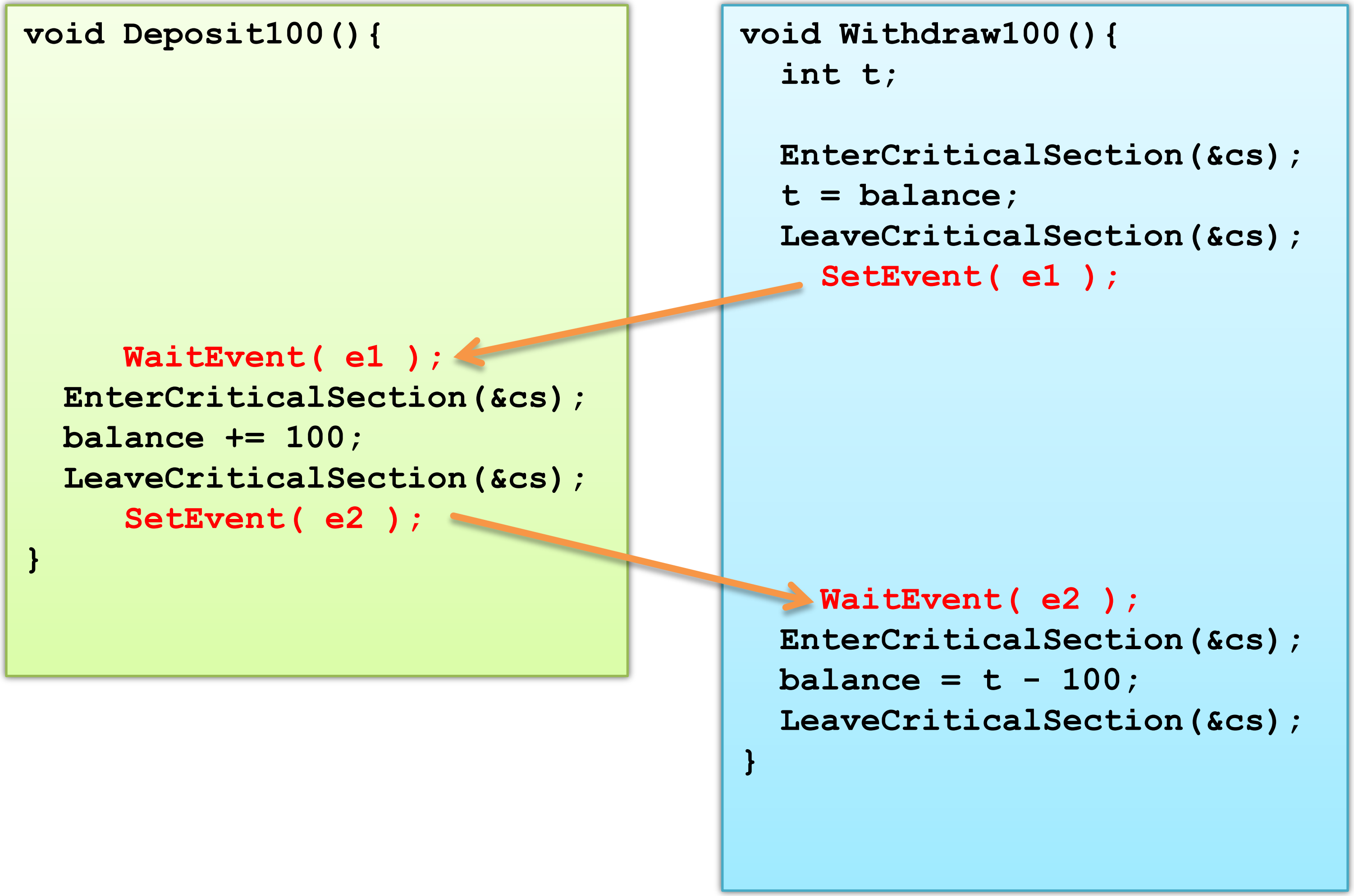
Deposit Thread

```
void Deposit100(){
  ChessSchedule();
  EnterCriticalSection(&cs);
  balance += 100;
  ChessSchedule();
  LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```
void Withdraw100(){
  int t;

  ChessSchedule();
  EnterCriticalSection(&cs);
  t = balance;
  ChessSchedule();
  LeaveCriticalSection(&cs);

  ChessSchedule();
  EnterCriticalSection(&cs);
  balance = t - 100;
  ChessSchedule();
  LeaveCriticalSection(&cs);

}
```

Introduce Predictable Delays with Additional Synchronization



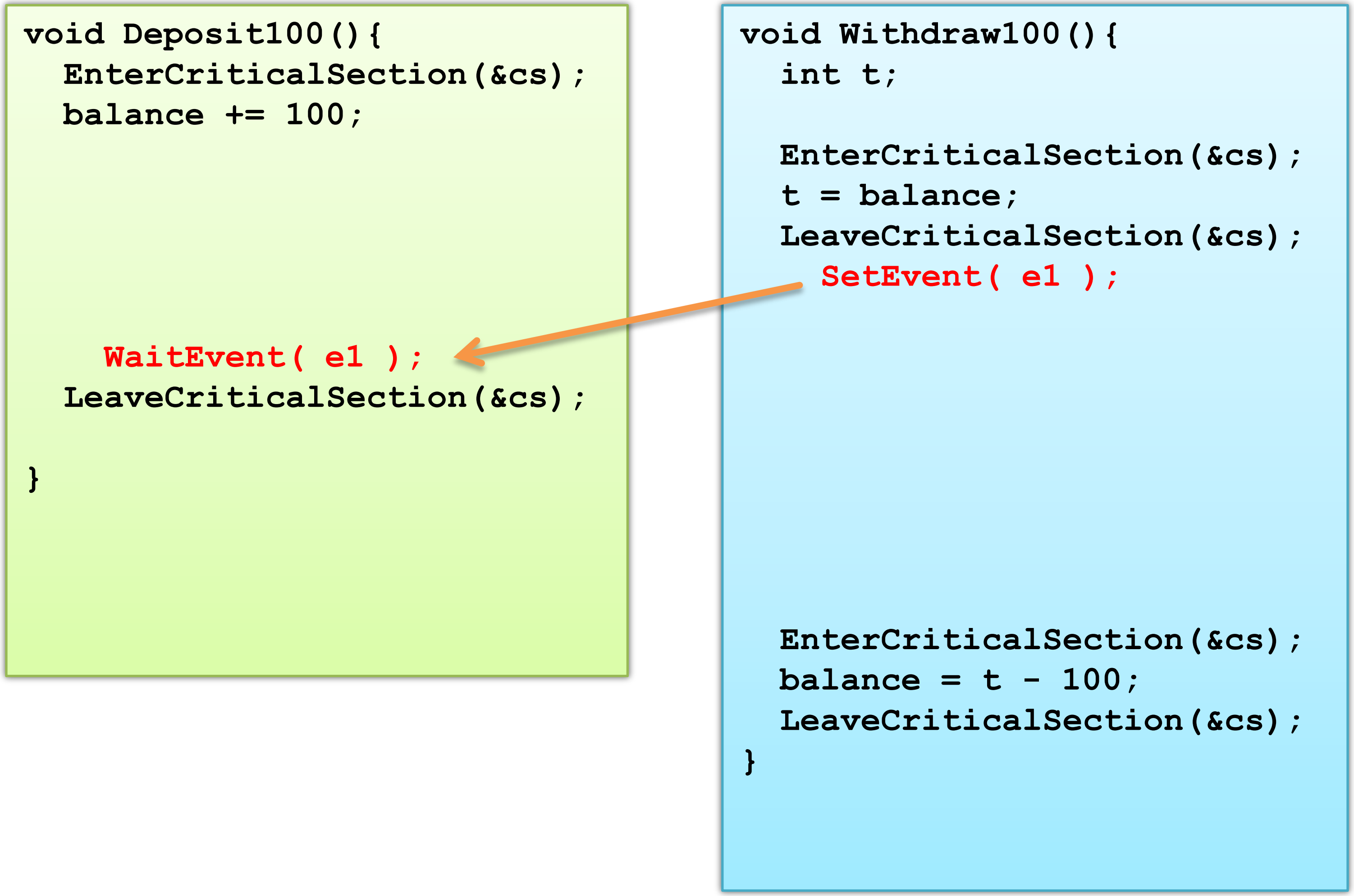
Deposit Thread

```
void Deposit100(){







    WaitEvent( e1 );
  EnterCriticalSection(&cs);
  balance += 100;
  LeaveCriticalSection(&cs);
    SetEvent( e2 );
}
```

Withdraw Thread

```
void Withdraw100(){
  int t;

  EnterCriticalSection(&cs);
  t = balance;
  LeaveCriticalSection(&cs);
    SetEvent( e1 );







    WaitEvent( e2 );
  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);
}
```

Blindly Inserting Synchronization Can Cause Deadlocks



Deposit Thread

```
void Deposit100(){
  EnterCriticalSection(&cs);
  balance += 100;






    WaitEvent( e1 );
  LeaveCriticalSection(&cs);

}
```

Withdraw Thread

```
void Withdraw100(){
  int t;

  EnterCriticalSection(&cs);
  t = balance;
  LeaveCriticalSection(&cs);
    SetEvent( e1 );








  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);
}
```

CHESS Scheduler Basics

- Introduce an event per thread
- Every thread blocks on its event
- The scheduler wakes one thread at a time by enabling the corresponding event
- The scheduler does not wake up a *disabled* thread
  - Need to know when a thread can make progress
  - Wrappers for synchronization provide this information
- The scheduler has to pick one of the enabled threads
  - The exploration engine decides for the scheduler

CHESS Synchronization Wrappers

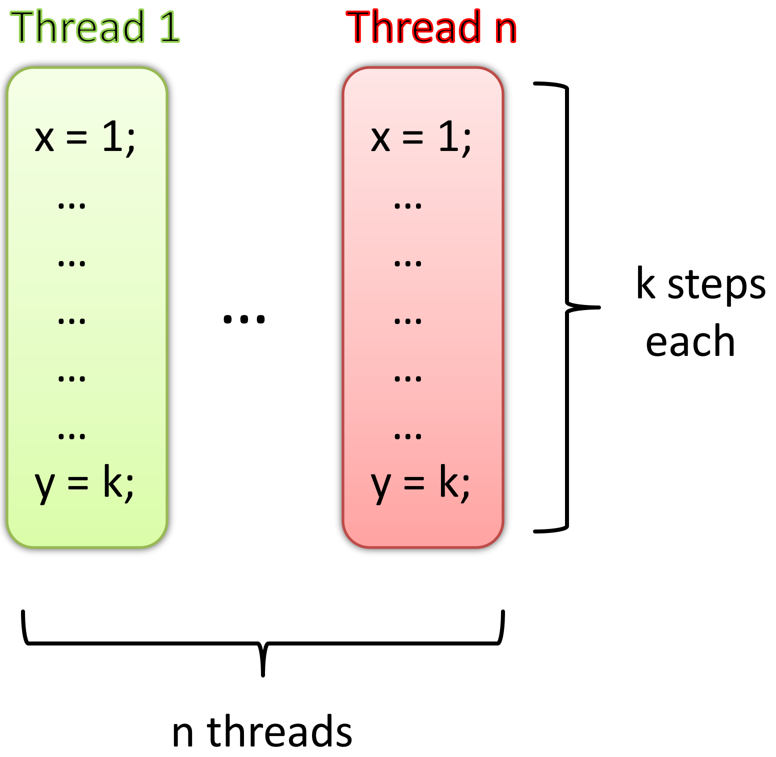
- Understand the semantics of synchronizations
- Provide enabled information

```
CHESS_EnterCS{
  while(true) {
    canBlock = TryEnterCS (&cs);
    if(canBlock)
       Sched.Disable(currThread);
  }
}
```

- Expose nondeterministic choices
    - An asynchronous ReadFile can possibly return synchronously

CHESS Algorithms

State space explosion



Thread 1

x = 1;
…
…
…
…
…
y = k;

…

Thread n

x = 1;
…
…
…
…
…
y = k;

k steps each

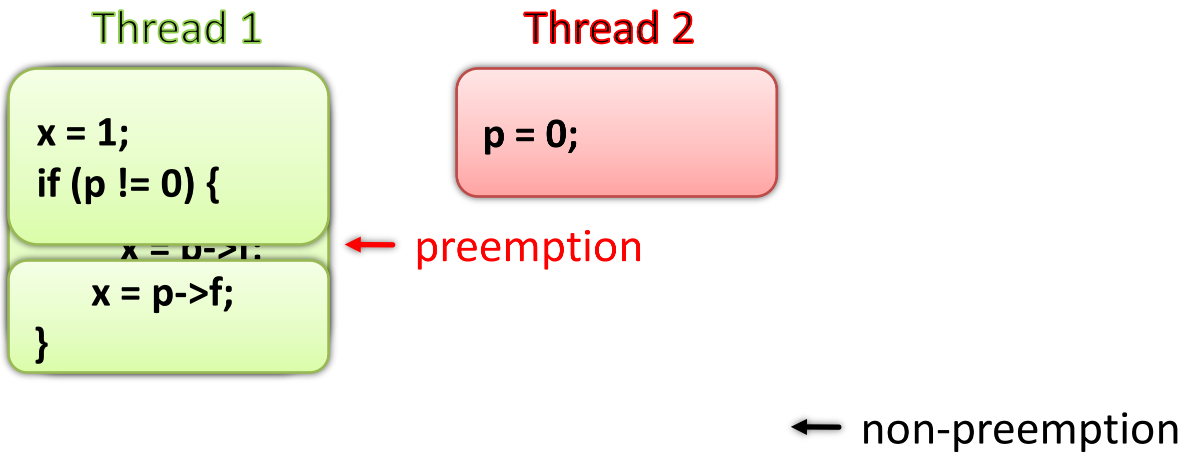
n threads

- 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

- CHESS, by default, is a non-preemptive, starvation-free scheduler
  - Execute huge chunks of code atomically
- Systematically insert a small number preemptions
  - Preemptions are context switches forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end



Thread 1

```
x = 1;
if (p != 0) {
    x = p->f;
}
```

Thread 2

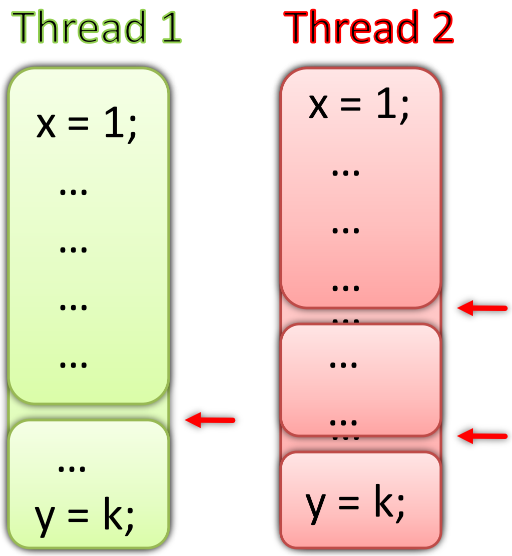
```
p = 0;
```

Polynomial state space

- Terminating program with fixed inputs and deterministic threads
  - n threads, k steps each, c preemptions
- Number of executions <= ${}_{nk}C_{c} \,.\, (n+c)!$

$$= O(\ (n^2k)^c .\ n!\ )$$

Exponential in n and c, but not in k



Thread 1

x = 1;
…
…
…
…
…
y = k;

Thread 2

x = 1;
…
…
…
…
…
…
…
y = k;

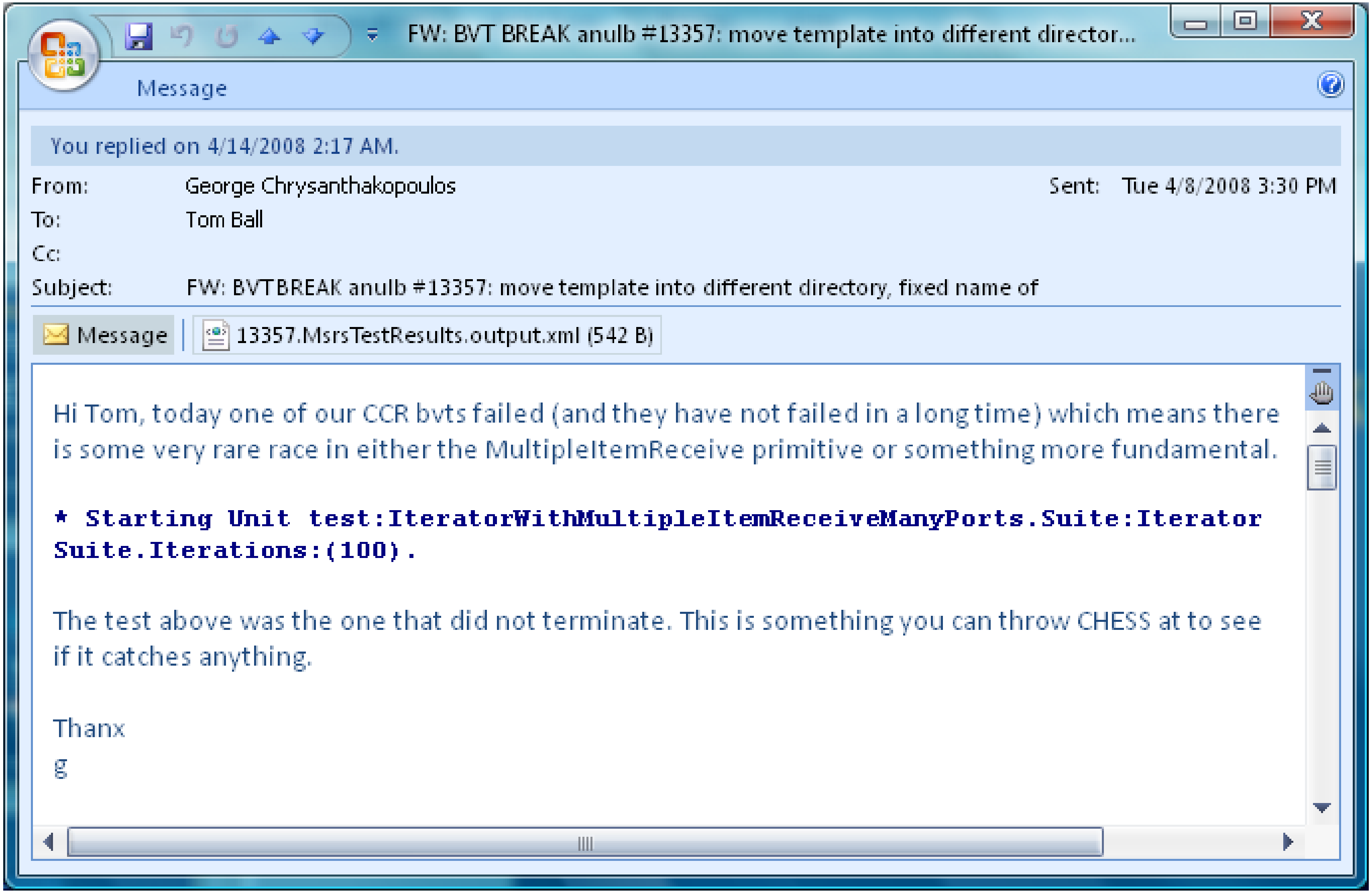
- Choose c preemption points
- Permute n+c atomic blocks

Advantages of preemption bounding

- Most errors are caused by few (<2) preemptions
- Generates an easy to understand error trace
  - Preemption points almost always point to the root-cause of the bug
- Leads to good heuristics
  - Insert more preemptions in code that needs to be tested
  - Avoid preemptions in libraries
  - Insert preemptions in recently modified code
- A good coverage guarantee to the user
  - When CHESS finishes exploration with 2 preemptions, any remaining bug requires 3 preemptions or more

Finding and reproducing CCR Heisenbug

George Chrysanthakopoulos' Challenge



You replied on 4/14/2008 2:17 AM.

From: George Chrysanthakopoulos
Sent: Tue 4/8/2008 3:30 PM
To: Tom Ball
Cc:
Subject: FW: BVTBREAK anulb #13357: move template into different directory, fixed name of

Hi Tom, today one of our CCR bvts failed (and they have not failed in a long time) which means there is some very rare race in either the MultipleItemReceive primitive or something more fundamental.

```
* Starting Unit test:IteratorWithMultipleItemReceiveManyPorts.Suite:Iterator
Suite.Iterations:(100).
```

The test above was the one that did not terminate. This is something you can throw CHESS at to see if it catches anything.

Thanx
g

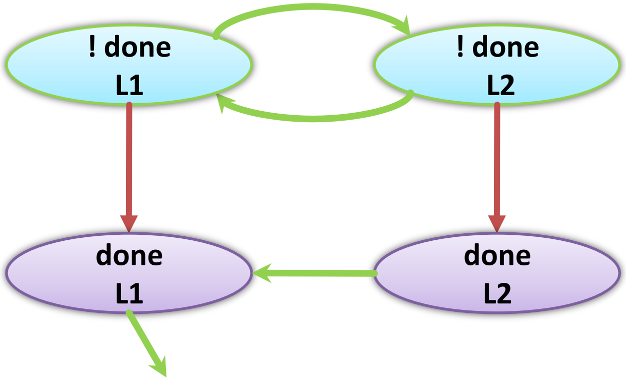
Concurrent programs have cyclic state spaces

Thread 1

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

Thread 2

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



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

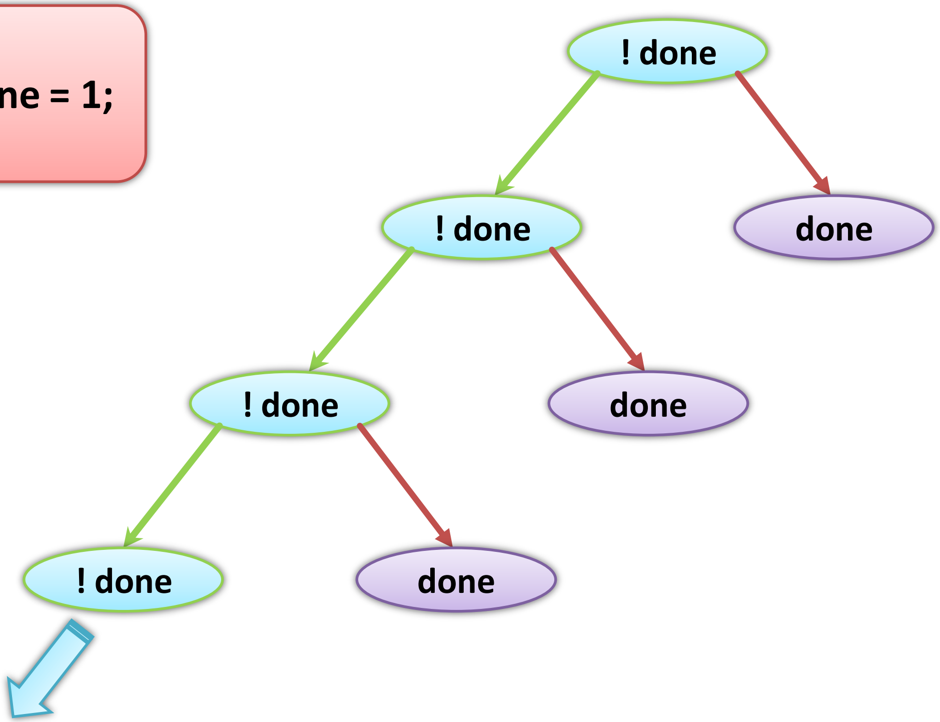
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

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

Thread 2

```
done = 1;
```



! done

! done

done

! done

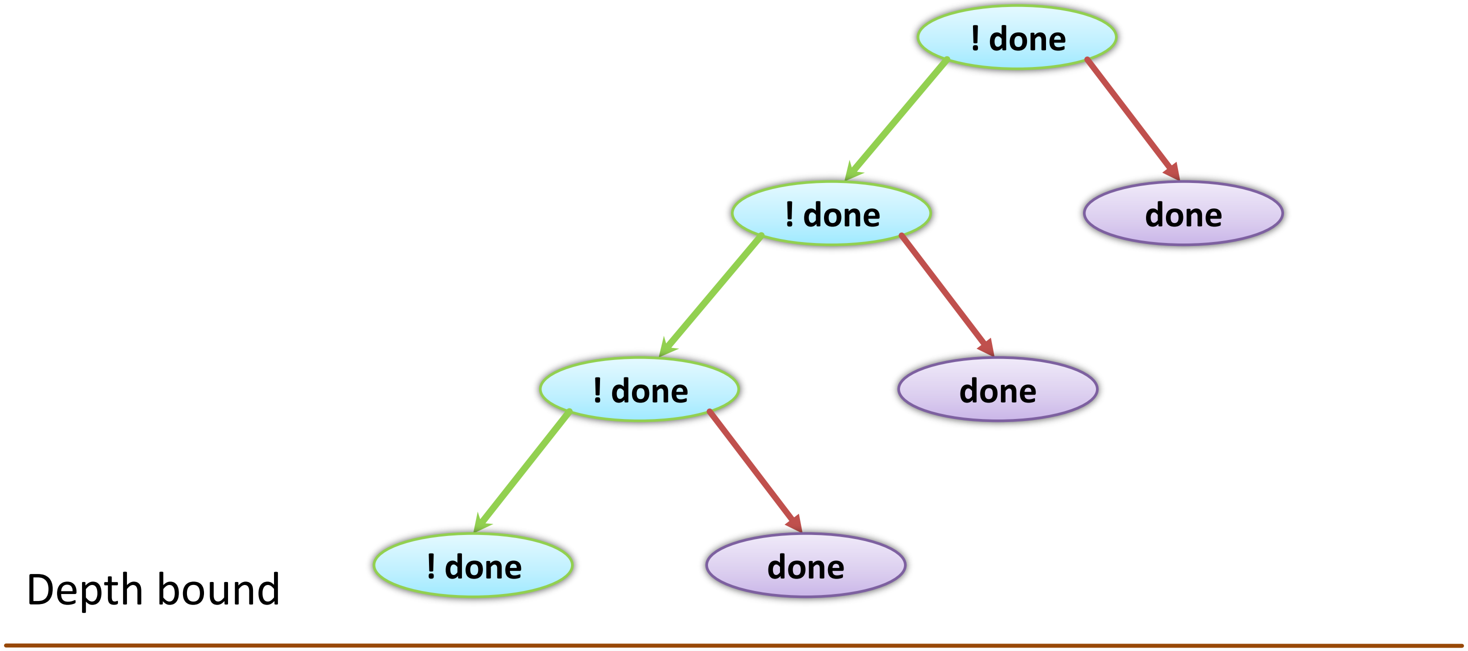
done

! done

done

Depth bounding

- Prune executions beyond a bounded number of steps



! done

! done

done

! done

done

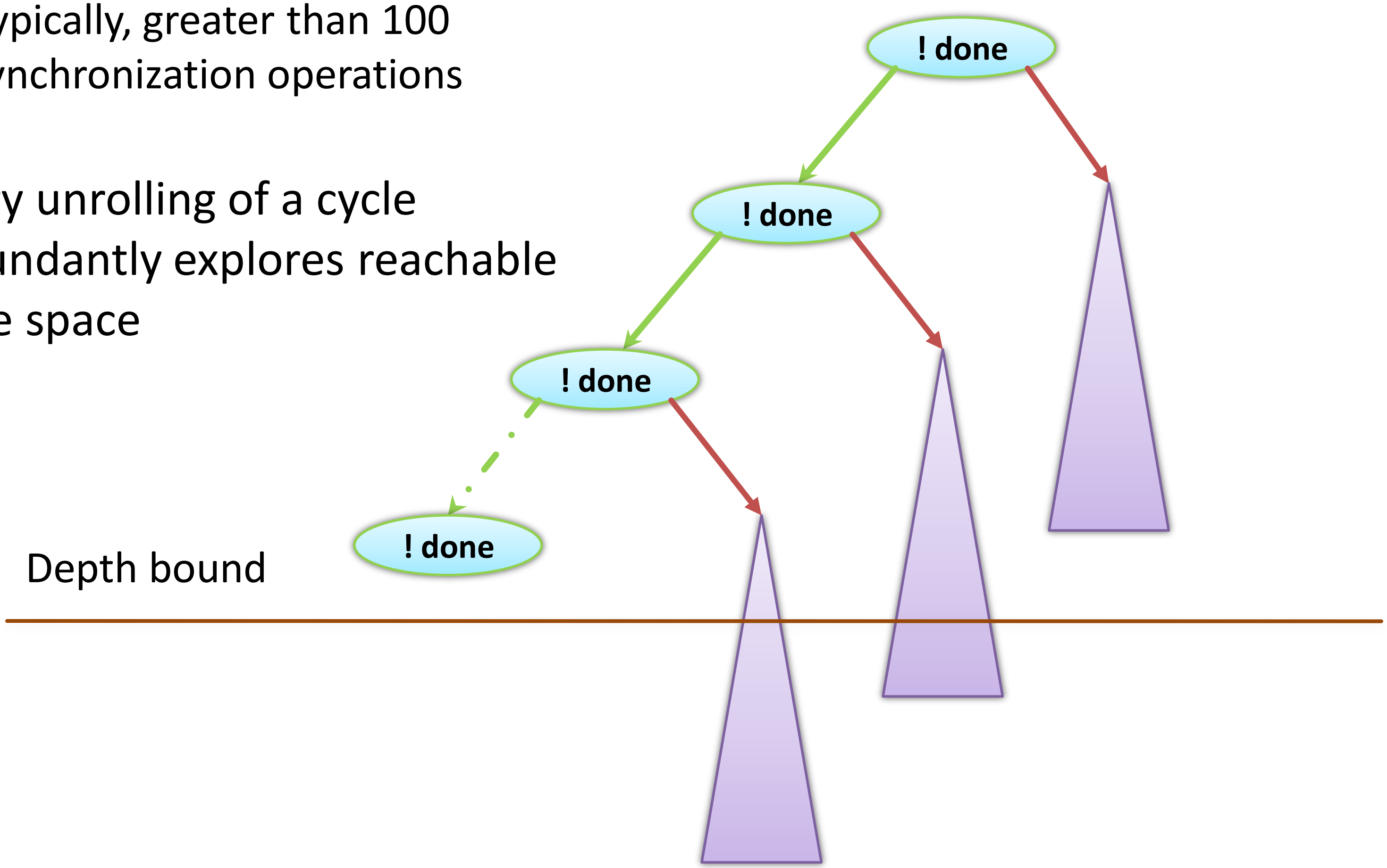
Depth bound

! done

done

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;
```

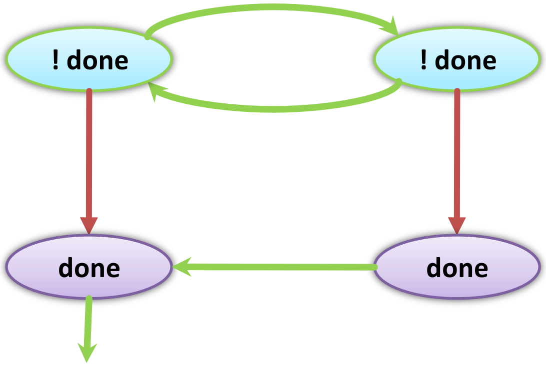
Key idea

Thread 1

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

Thread 2

```
done = 1;
```

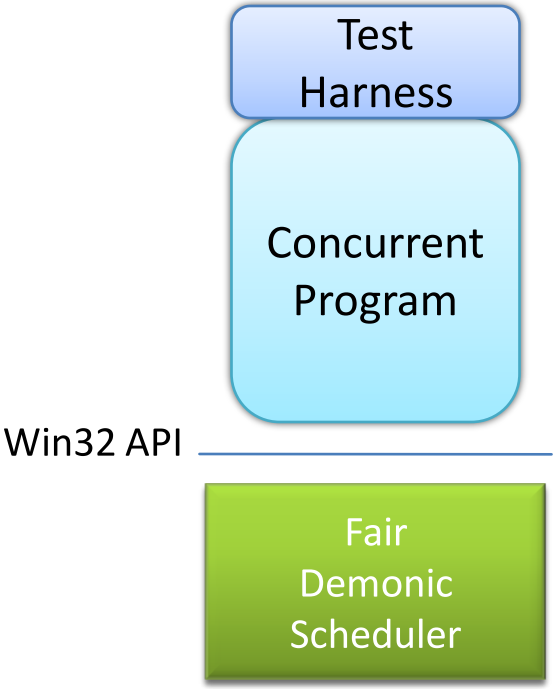


- This test terminates only when the scheduler is fair
- Fairness is assumed by programmers

All cycles in correct programs are unfair

A fair cycle is a livelock

We need a fair scheduler



- Avoid unrolling unfair cycles
  - Effective state coverage
- Detect fair cycles
  - Find livelocks

- What notion of “fairness” do we use?

Weak fairness

- Forall t :: GF (enabled(t) → scheduled(t))
- A thread that remains enabled should eventually be scheduled

Thread 1

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

Thread 2

```
done = 1;
```

- A weakly-fair scheduler will eventually schedule Thread 2
- Example: round-robin

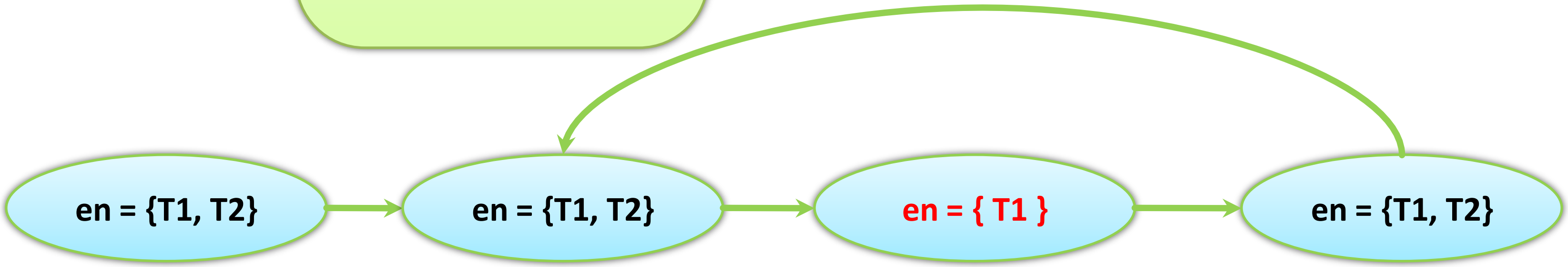
Weak fairness does not suffice

Thread 1

```
Lock( l );
While( ! done)
{
    Unlock( l );   ←
    Sleep();       ←
    Lock( l );     ←
}
Unlock( l );
```

Thread 2

```
Lock( l );  ←
done = 1;
Unlock( l );
```



T1: Sleep()
T2: Lock(l)

T1: Lock(l)
T2: Lock(l)

T1: Unlock(l)
T2: Lock(l)

T1: Sleep()
T2: Lock(l)

Strong Fairness

- Forall t :: GF enabled(t) → GF scheduled(t)
- A thread that is enabled infinitely often is scheduled infinitely often

Thread 1

```
Lock( l );
While( ! done)
{
      Unlock( l );
      Sleep();
      Lock( l );
}
Unlock( l );
```

Thread 2

```
Lock( l );
done = 1;
Unlock( l );
```

- Thread 2 is enabled and competes for the lock infinitely often

Implementing a strongly-fair scheduler

- Apt & Olderog ’83
  - A round-robin scheduler with priorities

- Operating system schedulers
  - Priority boosting of threads

We also need to be demonic

- Cannot generate all fair schedules
  - There are infinitely many, even for simple programs
- It is sufficient to generate enough fair schedules to
  - Explore all states (safety coverage)
  - Explore at least one fair cycle, if any (livelock coverage)
- Do it without capturing the program states

(Good) Programs indicate lack of progress

Thread 1

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

Thread 2

```
done = 1;
```

- Good Samaritan assumption:
  - Forall threads t : GF scheduled(t) → GF yield(t)
  - A thread when scheduled infinitely often yields the processor infinitely often
- Examples of yield:
  - Sleep(), ScheduleThread(), asm {rep nop;}
  - Thread completion

Robustness of the Good Samaritan assumption

- A violation of the Good Samaritan assumption is a performance error

Thread 1

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

Thread 2

```
done = 1;
```

- Programs are parsimonious in the use of yields
  - A Sleep() almost always indicates a lack of progress
  - Implies that the thread is stuck in a state-space cycle

Fair demonic scheduler

- Maintain a priority-order (a partial-order) on threads
  - t < u : t will not be scheduled when u is enabled
- Threads get a lower priority only when they yield
  - Scheduler is fully demonic on yield-free paths
  - When t yields, add t < u if
    - Thread u was continuously enabled since last yield of t, or
    - Thread u was disabled by t since the last yield of t
- A thread loses its priority once it executes
  - Remove all edges t < u when u executes

Four outcomes of the semi-algorithm

- Terminates without finding any errors
- Terminates with a safety violation
- Diverges with an infinite execution
  - that violates the GS assumption (a performance error)
  - that is strongly-fair (a livelock)

- In practice: detect infinite executions by a very long execution

Data Races & Memory Model Races

What is a Data Race?

- 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

- Best practice: write “correctly synchronized“ programs that do not contain data races.

What Makes Data Races significant?

- Data races may reveal synchronization errors
  - Most typically, programmer forgot to take a lock, use an interlocked operation, or declare a variable volatile.
  - Racy programs risk obscure failures caused by memory model relaxations in the hardware and the compiler
  - But: many programmers tolerate “benign” races

- Race-free programs are easier to verify
  - if program is race-free, it is enough to consider schedules that preempt on synchronizations only
  - CHESS heavily relies on this reduction

How do we find races?

- Remember: races are concurrent conflicting accesses.
- But what does concurrent actually mean?
- Two general approaches to do race-detection

Lockset-Based
(heuristic)
Concurrent ≈
“Disjoint locksets”

Happens-Before-Based
(precise)
Concurrent =
“Not ordered by happens-before”

Synchronization = Locks ???

- This C# code contains neither locks nor a data race:

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

Thread 1

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

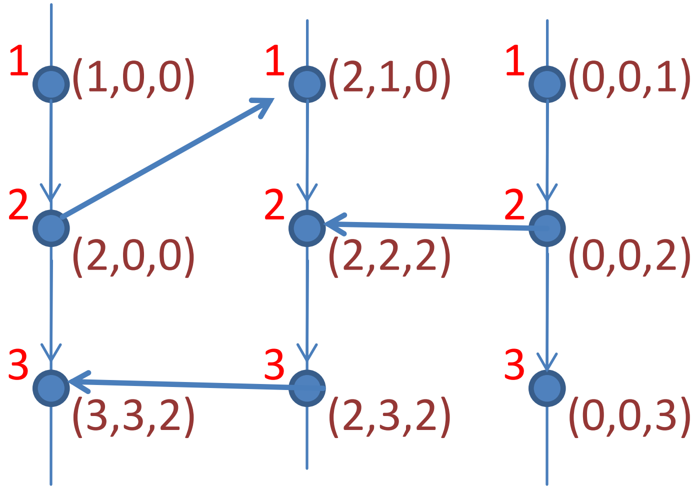
Thread 2

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

- CHESS is *precise*: does not report this as a race. But *does* report a race if you remove the ‘volatile’ qualifier.

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)
    - Etc.

Example

Static Program

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

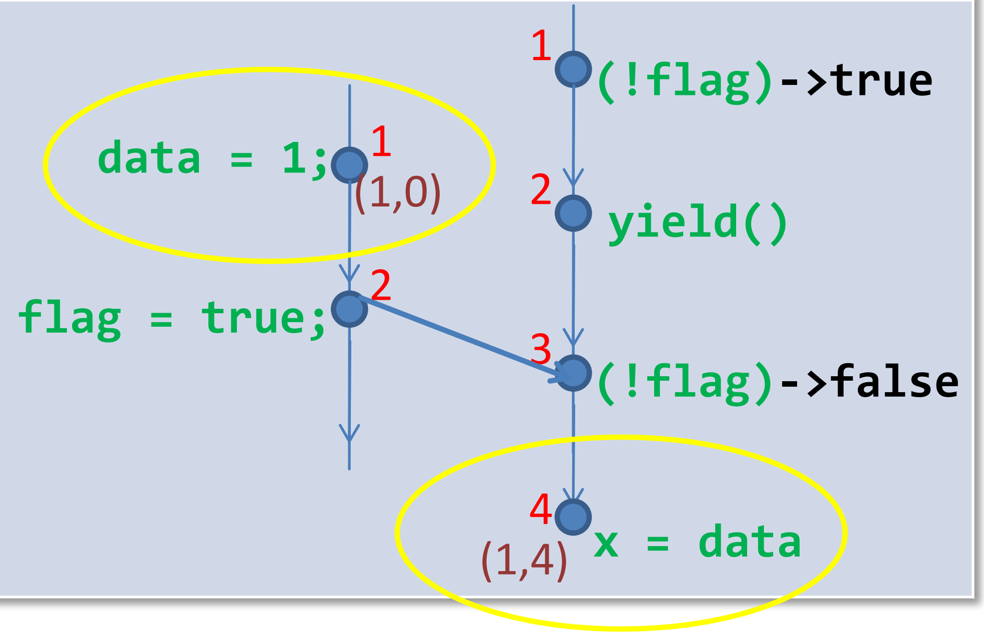
Thread 1

```
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.

Basic Algorithm

- For each explored schedule,
  - Execute code and timestamp all data accesses.
  - Check if there were any conflicting concurrent accesses to some location.

- This basic algorithm can be optimized in many ways
  - On-the-fly checking, Memory management
  - Lightweight alternatives to full vector clocks
  - See [Flanagan PLDI 09]

Reduction for Race-Free Programs

- By default, CHESS preempts on synchronization accesses only
  - May miss bugs if program contains data race

- If we turn on race detection, CHESS can verify that the reduction is sound by verifying absence of data races.

- Thus, for race-free programs, we get both:
  - Full guarantee
  - Reduction in the number of schedules

Preemption / Instrumentation Level

- Speed/coverage tradeoff : choose mode

	Sync only	**Sync. + vol. (Default)**	**Sync + vol. + Race Detection**	**All accesses**
Locks, Events, Interlocked, etc.	Instrumented & Preempted	Instrumented & Preempted	Instrumented & Preempted	Instrumented & Preempted
Volatile Accesses	-	Instrumented & Preempted	Instrumented & Preempted	Instrumented & Preempted
All Data Accesses	-	-	Instrumented	Instrumented & Preempted

Demos: SimpleBank / CCR

- Find a simple data race in a toy example
- Find a not-so-simple data race in production code

Bugs Caused By Relaxed Memory Models

- Programmers avoid locks in performance-critical code
  - Faster to use normal loads and stores, or interlocked operations
- Low-lock code can break on **relaxed memory models**
  - Most multicore machines (including x86) do not guarantee sequential consistency of memory accesses

- Vulnerabilities are hard to find, reproduce, and analyze
  - Show up only on multiprocessors
  - Often not reproduceable

Example: Store Buffers Break Dekker

- On an ideal (sequentially consistent) multiprocessor, this code never executes foo() and bar() at the same time

```
volatile int A;
volatile int B;

 Thread 1            Thread 2
 ---------           ---------
 A = 1;              B = 1;
 If (B == 0)         If (A == 0)
   foo();              bar();
```

- But on x86 (and almost all other multiprocessors), it may, because of store buffers.

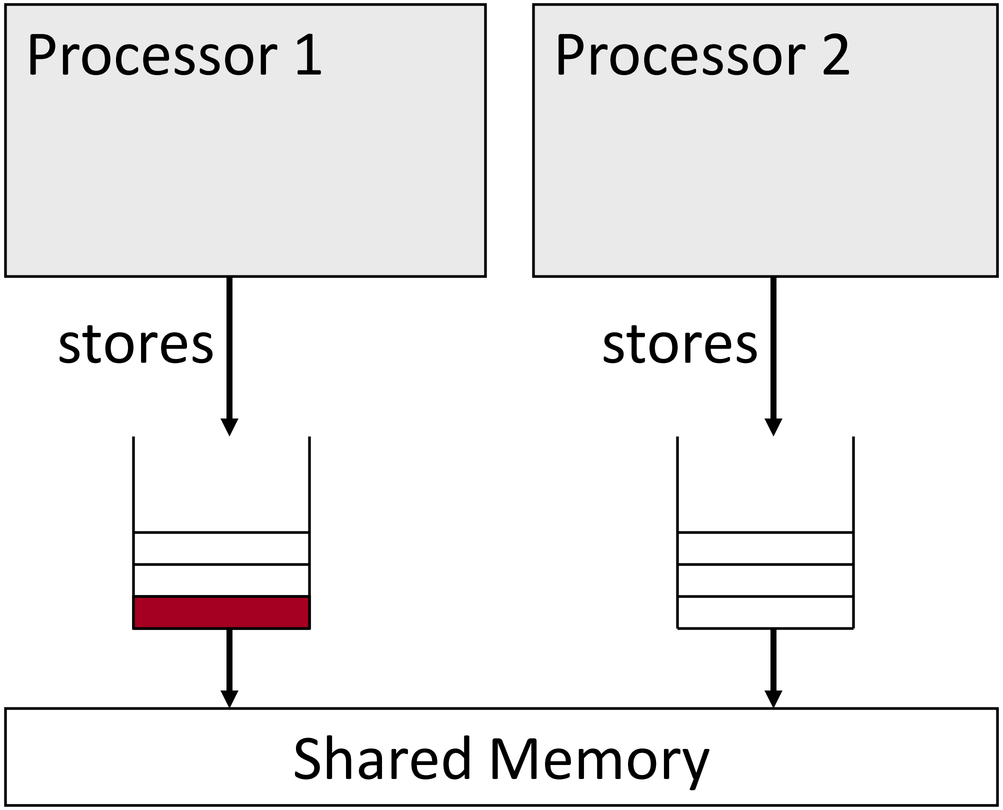
Memory Access Terminology

C++	Java	C#
atomic	volatile	interlocked
low-level atomic	-	volatile
volatile	-	-
(regular)	(regular)	(regular)

- Code using accesses marked red for synchronization purposes is susceptible to store buffer bugs.

Store Buffers

- Each processor buffers its own writes in a FIFO store buffer
- Remote processors do not see the buffered write until it is committed to shared memory
- Local processor "snoops" its own buffer when reading from memory
- Important for hardware performance



How to Find Store Buffer Bugs?

- Naïve: simulate machine
  - Too many schedules.
- Better: build a *borderline monitor* [CAV 2008].

Idea: While exploring schedules under CHESS, check for *stale loads*.

- A *stale load* is a load that may return a value under TSO that it could never return under SC.
- [Thm.] A program is TSO-safe if and only if all executions are free of stale loads.

Demos: Dekker / PFX

- Basic test: Dekker
- Found 2 dekker-like synchronization errors in production code
  - “optimization” of signal-wait pattern
  - Double-ended work-stealing queue

```
volatile bool isIdling;
volatile bool hasWork;

   //Consumer thread
   void BlockOnIdle(){
      lock (condVariable){
         isIdling = true;
         if (!hasWork)
             Monitor.Wait(condVariable);
         isIdling = false;
      }
   }

   //Producer thread
   void NotifyPotentialWork(){
      hasWork = true;
      if (isIdling)
        lock (condVariable) {
           Monitor.Pulse(condVariable);
        }
   }
```

Store Buffer Bugs - Experience

- Relatively rare… found only 3 so far
  - We expect to find more as we cover more code… detection is on by default whenever race detection is on
  - Found 1 false positive so far (i.e. “benign” stale load).

- Very common for certain algorithms, e.g. work stealing queue
  - We found one in PFX work-stealing queue
  - Know of 4 other teams (inside & outside Microsoft) who faced store buffer issues when implementing work-stealing queue

Writing a CHESS Monitor

Specifications?

- We have not seen significant practical success of verification methodology that requires extensive formal specification.

- More pragmatic: monitor certain or likely indicators automatically. Currently, we…
  - …flag error on: Deadlock, Livelock, Assertion Violation.
  - …generate warnings for: Data races, Stale loads.

More Monitors Find More Bugs

- Use runtime monitors for ‘typical programmer mistakes’
  - Data Races, Stale Loads (✓)
  - Atomicity violations, High-level Data Races
  - Incorrect API usage (for all kinds of APIs), e.g. Memory Leaks
- Much existing research on runtime monitors
- **CHESS SDK** provides infrastructure, you write your own monitor.

Monitors Benefit from Infrastructure

- Instrumentation
  - For both C# and C/C++
- Abstraction
  - Threads, synchronization & data variables, events
- Sequential schedule
  - Monitors need not worry about concurrent callbacks
- Repro capability
  - Any errors found can be reproduced deterministically
- Schedule enumeration
  - Enumerates schedules using reductions & heuristics
  - turns runtime monitors into verification tools

Chess <-> Monitor interface

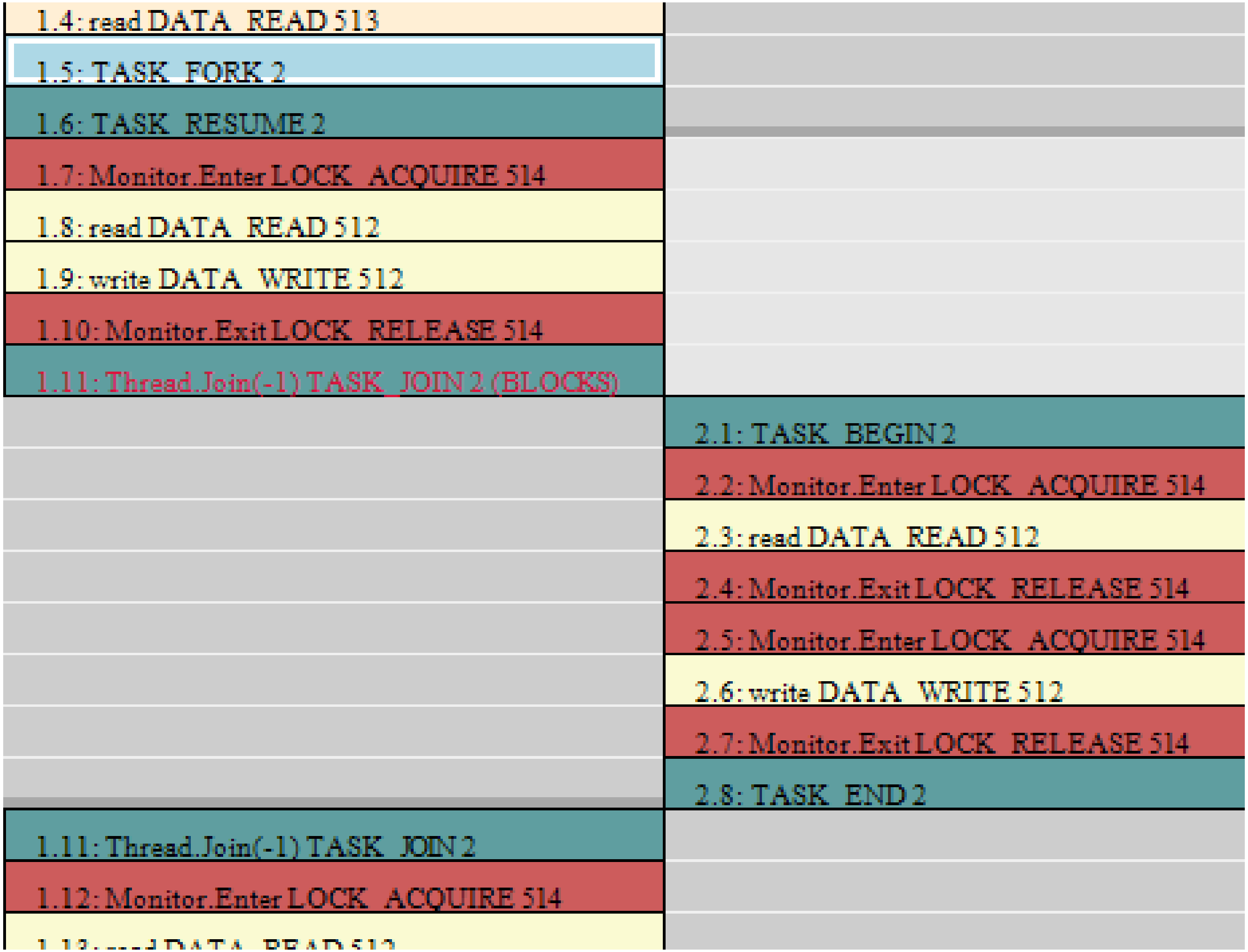
- Each monitor gets called by CHESS repeatedly
  - … at beginning and end of each schedule
  - … on relevant program events
    - Synchronization operations
    - Data variable accesses
    - User-defined instrumentation

- Callbacks abstract many low-level details
  - Handle plethora of synchronization APIs and concurrency constructs under the covers

Abstractions Provided

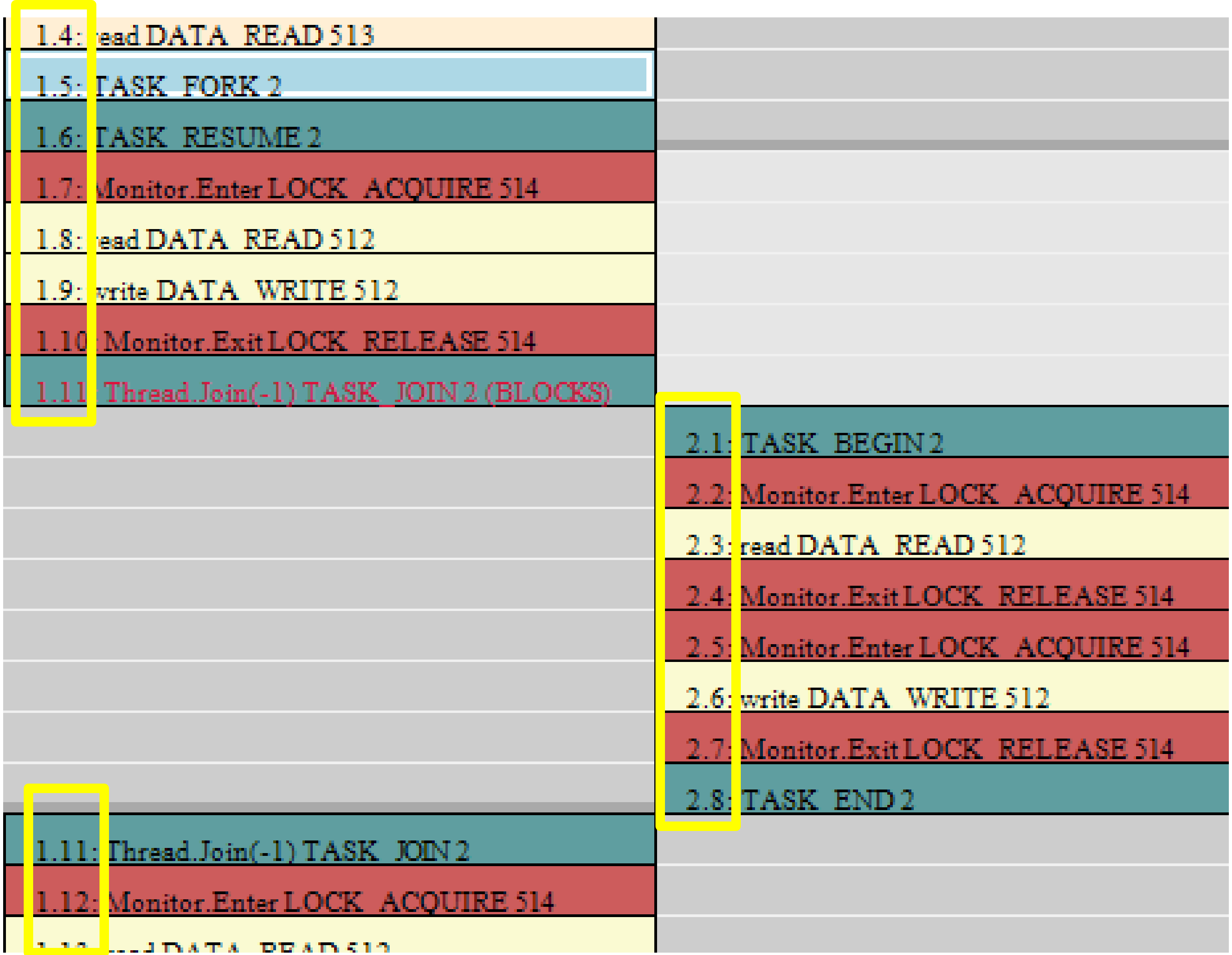
- Thread id = integer
  - Chess numbers threads consecutively 1, 2, 3, ….
- Event id = integer x integer
  - Chess numbers events in each thread consecutively 1.1, 1.2, 1.3, …. 2.1., 2.2., 2.3, …
- Syncvar = integer
  - Abstractly represents a synchronization object (lock, volatile variable, etc.)
- SyncvarOp = { LOCK_ACQUIRE, LOCK_RELEASE, RWVAR_READWRITE, RWVAR_READ, RWVAR_WRITE, TASK_FORK, TASK_JOIN, TASK_START, TASK_RESUME, TASK_END, …}
  - Represents synchronization operation on syncvar

ConcurrencyExplorer View of Schedule



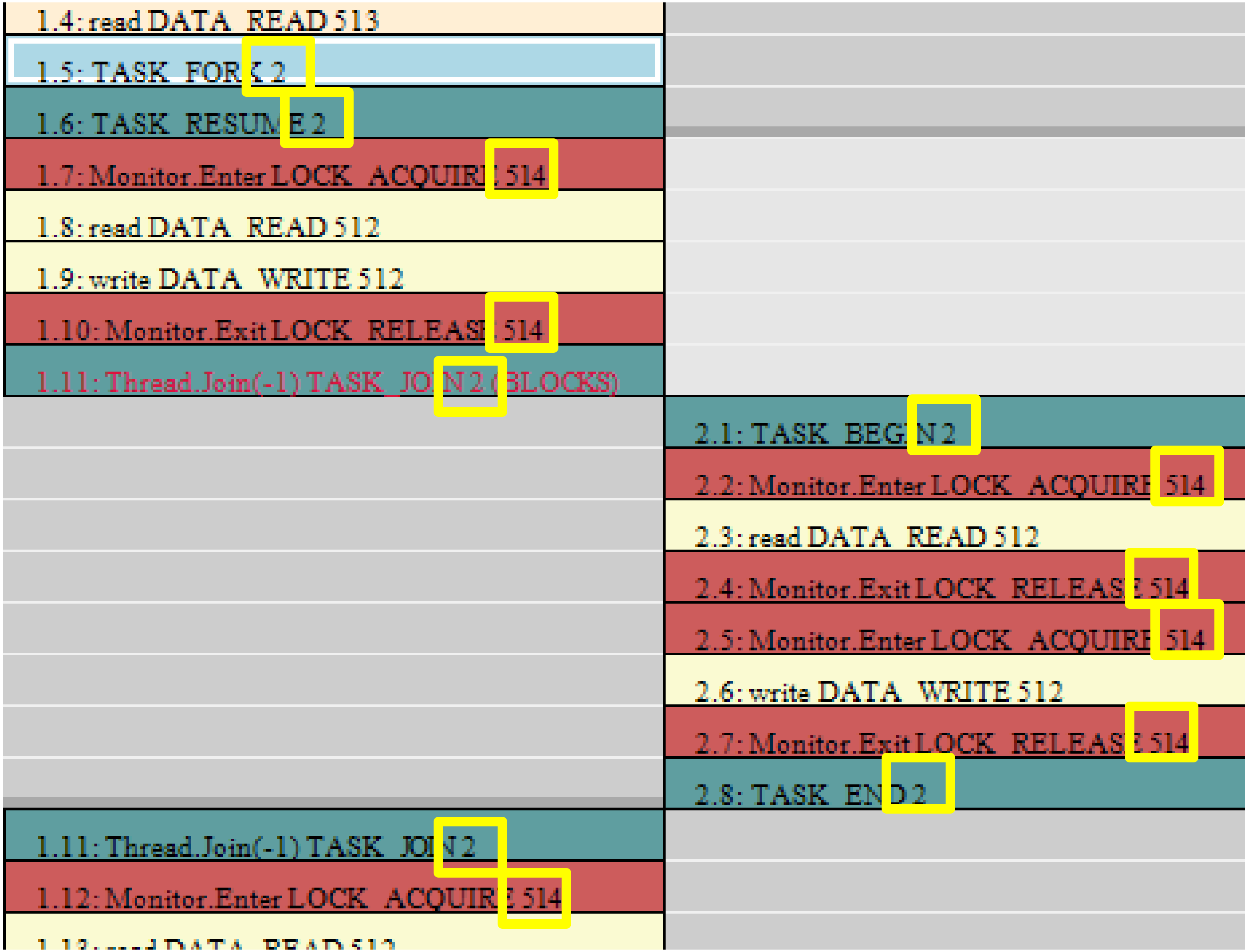
1.4: read DATA_READ 513	
1.5: TASK_FORK 2	
1.6: TASK_RESUME 2	
1.7: Monitor.Enter LOCK_ACQUIRE 514	
1.8: read DATA_READ 512	
1.9: write DATA_WRITE 512	
1.10: Monitor.Exit LOCK_RELEASE 514	
1.11: Thread.Join(-1) TASK_JOIN 2 (BLOCKS)	
	2.1: TASK_BEGIN 2
	2.2: Monitor.Enter LOCK_ACQUIRE 514
	2.3: read DATA_READ 512
	2.4: Monitor.Exit LOCK_RELEASE 514
	2.5: Monitor.Enter LOCK_ACQUIRE 514
	2.6: write DATA_WRITE 512
	2.7: Monitor.Exit LOCK_RELEASE 514
	2.8: TASK_END 2
1.11: Thread.Join(-1) TASK_JOIN 2	
1.12: Monitor.Enter LOCK_ACQUIRE 514	
1.13: read DATA_READ 512	

Event IDs



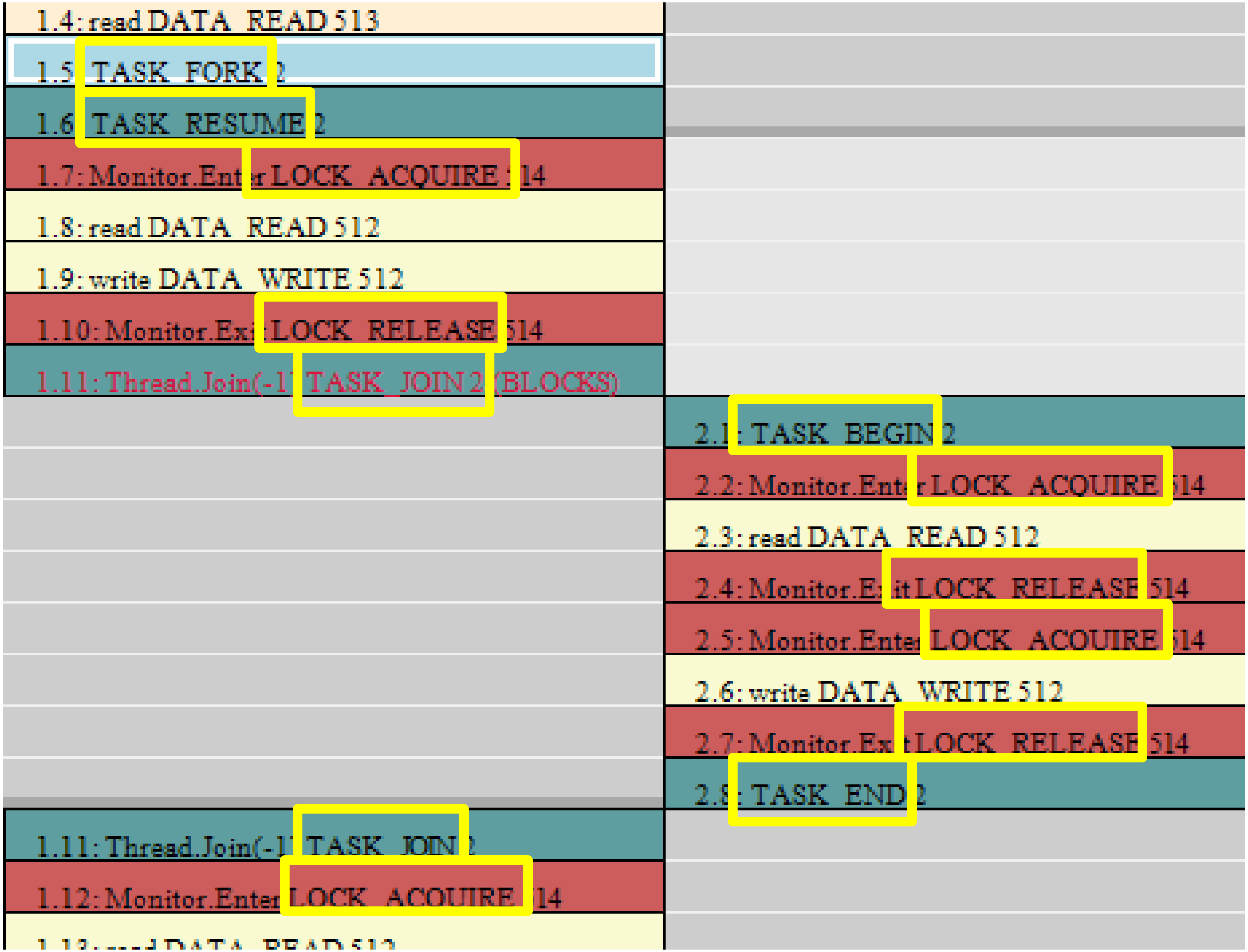
1.4: read DATA_READ 513	
1.5: TASK_FORK 2	
1.6: TASK_RESUME 2	
1.7: Monitor.Enter LOCK_ACQUIRE 514	
1.8: read DATA_READ 512	
1.9: write DATA_WRITE 512	
1.10: Monitor.Exit LOCK_RELEASE 514	
1.11: Thread.Join(-1) TASK_JOIN 2 (BLOCKS)	
	2.1: TASK_BEGIN 2
	2.2: Monitor.Enter LOCK_ACQUIRE 514
	2.3: read DATA_READ 512
	2.4: Monitor.Exit LOCK_RELEASE 514
	2.5: Monitor.Enter LOCK_ACQUIRE 514
	2.6: write DATA_WRITE 512
	2.7: Monitor.Exit LOCK_RELEASE 514
	2.8: TASK_END 2
1.11: Thread.Join(-1) TASK_JOIN 2	
1.12: Monitor.Enter LOCK_ACQUIRE 514	

SyncVar



1.4: read DATA_READ 513	
1.5: TASK_FORK 2	
1.6: TASK_RESUME 2	
1.7: Monitor.Enter LOCK_ACQUIRE 514	
1.8: read DATA_READ 512	
1.9: write DATA_WRITE 512	
1.10: Monitor.Exit LOCK_RELEASE 514	
1.11: Thread.Join(-1) TASK_JOIN 2 (BLOCKS)	
	2.1: TASK_BEGIN 2
	2.2: Monitor.Enter LOCK_ACQUIRE 514
	2.3: read DATA_READ 512
	2.4: Monitor.Exit LOCK_RELEASE 514
	2.5: Monitor.Enter LOCK_ACQUIRE 514
	2.6: write DATA_WRITE 512
	2.7: Monitor.Exit LOCK_RELEASE 514
	2.8: TASK_END 2
1.11: Thread.Join(-1) TASK_JOIN 2	
1.12: Monitor.Enter LOCK_ACQUIRE 514	
1.13: read DATA_READ 512	

SyncVarOp



1.4: read DATA_READ 513
1.5: TASK_FORK 2
1.6: TASK_RESUME 2
1.7: Monitor.Enter LOCK_ACQUIRE 514
1.8: read DATA_READ 512
1.9: write DATA_WRITE 512
1.10: Monitor.Exit LOCK_RELEASE 514
1.11: Thread.Join(-1) TASK_JOIN 2 (BLOCKS)
2.1: TASK_BEGIN 2
2.2: Monitor.Enter LOCK_ACQUIRE 514
2.3: read DATA_READ 512
2.4: Monitor.Exit LOCK_RELEASE 514
2.5: Monitor.Enter LOCK_ACQUIRE 514
2.6: write DATA_WRITE 512
2.7: Monitor.Exit LOCK_RELEASE 514
2.8: TASK_END 2
1.11: Thread.Join(-1) TASK_JOIN 2
1.12: Monitor.Enter LOCK_ACQUIRE 514
1.13: read DATA_READ 512

Some Callbacks

- At beginning & end of schedule

```
virtual void OnExecutionBegin(IChessExecution* exec)
virtual void OnExecutionEnd(IChessExecution* exec)
```

- Right after a synchronization operation:

```
virtual void OnSyncVarAccess(EventId id, Task tid,
                              SyncVar var, SyncVarOp op, size_t sid)
```

- Right after a data access:

```
virtual void OnDataVarAccess(EventId id, void* loc, int
                              size, bool isWrite, size_t pcId)
```

- Right before a synchronization operation:

```
virtual void OnSchedulePoint(EventId id, SyncVar var,
                                       SyncVarOp op, size_t sid)
```

Happens-before information

- Can query ‘character’ of a sync var op

```
static bool IsWrite(SyncVarOp op)
static bool IsRead(SyncVarOp op)
```

- Get happens-before edges between two sync-var ops
  - To the same variables
  - At least one of which is a write
- Note: most syncvarops are considered to be both reads & writes

Reduction-Compatible Monitors

- Different schedules may produce same hb-execution
  - Call such schedules hb-equivalent
- Program behaves identically under hb-equivalent schedules
  - Thus, reductions are sound (sleep-sets, data-race-free)
- But: some monitors may not behave equivalently
  - E.g. naïve race detection may require specific schedule
  - For coverage guarantees, monitor must be reduction-compatible: must detect error on all hb-equivalent schedules
- Our Race Detection and Store Buffer Detection are Reduction -Compatible

Refinement Checking

Concurrent Data Types

- Frequently used building blocks for parallel or concurrent applications.
- Typical examples:
  - Concurrent stack
  - Concurrent queue
  - Concurrent deque
  - Concurrent hashtable
  - ….
- Many slightly different scenarios, implementations, and operations
- Written by experts… but the experts need help

Correctness Criteria

- Say we are verifying concurrent X
  (for X $\in$ queue, stack, deque, hashtable …)
- Typically, concurrent X is expected to behave like atomically interleaved sequential X
- We can check this without knowing the semantics of X
- Implement easy to use, automatic consistency check

Observation Enumeration Method

[CheckFence, PLDI07]

- Given concurrent test, e.g.

```
          Stack s = new ConcurrentStack();
s.Push(1);       b1 = s.Pop(out i1);
                 b2 = s.Pop(out i2);
```

- (Step 1 : Enumerate Observations)
  Enumerate coarse-grained interleavings and record observations
  1. b1=true i1=1 b2=false i2=0
  2. b1=false i1=0 b2=true i2=1
  3. b1=false i1=0 b2=false i2=0
- (Step 2 : Check Observations)
  Check refinement: all concurrent executions must look like one of the recorded observations

Demo

- Show refinement checking on simple stack example

Conclusion

- CHESS is a tool for
  - Systematically enumerating thread interleavings
  - Reliably reproducing concurrent executions
- Coverage of Win32 and .NET API
  - Isolates the search & monitor algorithms from their complexity
- CHESS is extensible
  - Monitors for analyzing concurrent executions
  - Future: Strategies for exploring the state space