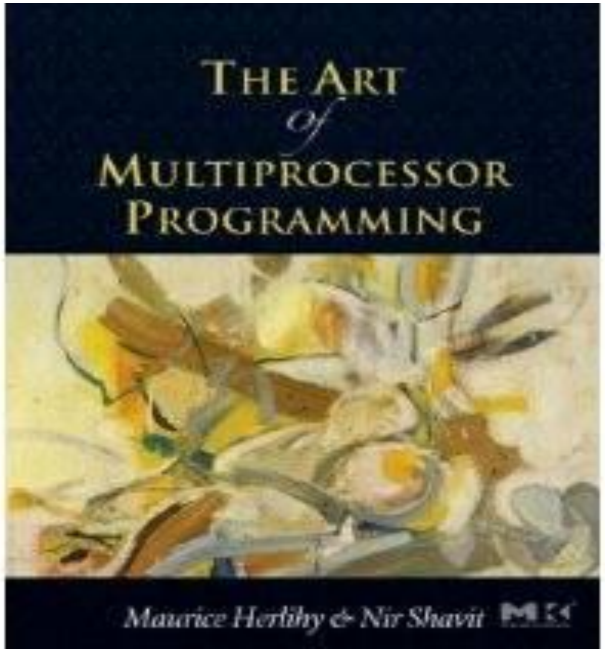
CS331 Tutorial 3
Java Thread Programming
(Locking and CDS)

A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Some slides are adopted from "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit



Outline

- Java Threading Examples
- Java Incrementing Counter: Multithreaded Version
- Locking in Java
- Concurrent Data Structure
  - List, Hash, Queue, Stack
- Simulation of PDES

Thread Programing In Java

Create Java Thread: Runnable Interface

```
public class MyThIR implements Runnable{
      public void run(){
            System.out.println("Child
                  thread is running...");
      }
}
public static void main(String args[]){
      MyThIR  m1=new  MyThIR();
      Thread t1 =new Thread(m1);
      t1.start();
 }
```

Create Java Thread: Extending Java Thread

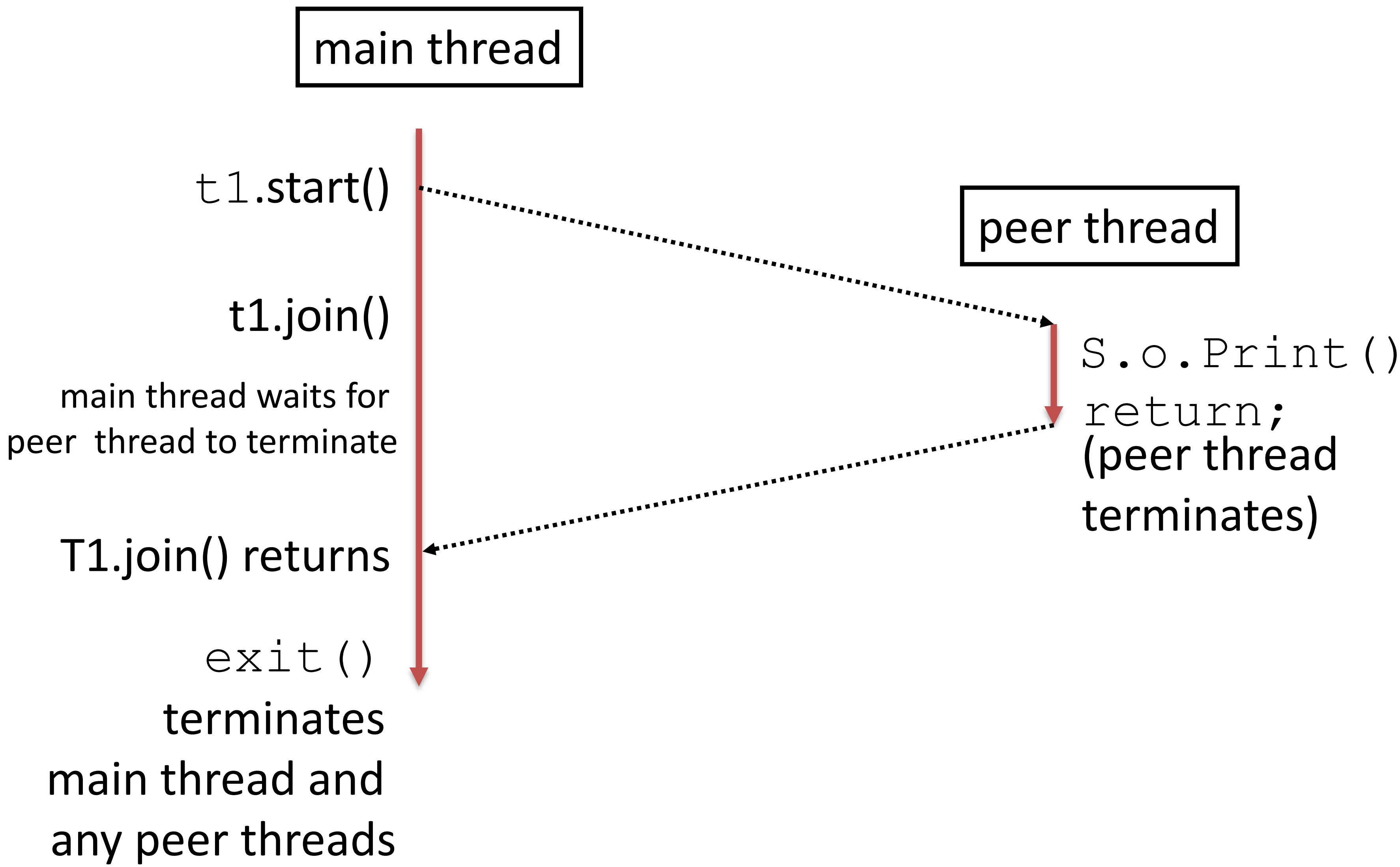
```
public class MyTh_EJT extends Thread{
      public void run(){
            System.out.println("Child
                  thread is running...");
      }
}

public static void main(String args[]){
      MyTh_EJT  t1 =new  MyTh_EJT();
      t1.start();
}
```

Create Java Thread: Extending Java Thread

```
public class MyTh_EJT extends Thread{
      public void run(){
            System.out.println("Child
                  thread is running...");
            return;
      }
}
public static void main(String args[]){
      MyTh_EJT  t1 =new  MyTh_EJT();
      t1.start();
      t1.join();
}
```

Execution of Threaded “child thread”



VectorSum Serial

```
int A[VSize], B[VSize], C[VSize];
void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

Suppose Size=1000

0-249	250-499	500-749	750-999
T1	T2	T3	T4

VectorSum Serial

```
int A[VSize], B[VSize], C[VSize];
void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

- Independent
- Divide work into equal for each thread
- Work per thread: Size/numThread

VectorSum Parallel

```
void DoVectorSum(int TID){
    int j, SzPerthrd, LB, UB;
     SzPerthrd=(VSize/NUM_THREADS);
     LB= SzPerthrd*TID;UB=LB+SzPerthrd;

    for(j=LB;j<UB;j++)
     A[j]=B[j]+C[j];
}
```

VectorSum Parallel : Java Threads

```
public class MyTh_EJT extends Thread{
      public void run(int TID){
            DoVectorSum(TID)
            return;
      }
}
public static void main(String args[]){
   int NumOfTasks = Integer.parseInt(args[0]);
   for ( int i=0; i < NumOfTasks; i++) {
        MyTh_EJT t1=new  MyTh_EJT();
        t1.start(i);
      }
}
```

VectorSum Parallel : Java Threads

Complete java program will be uploaded to MS Team

Four ways to implement a synchronized counter in Java

- Suppose there is a Shared counter and every threads are attempting to manipulate
  1. Synchronized Block
  2. Atomic Variable
  3. Concurrent Lock
  4. Semaphore

0: Simple Java Ctr: without Lock

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Counter implements Runnable {
    private static int counter = 0;
    private static final int limit = 1000;
    private static final int threadPoolSize = 5;

   public static void main(String[] args) {
        ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
        for (int i = 0; i < threadPoolSize; i++) { ES.submit(new Counter());   }
        ES.shutdown();
    }
}
```

0: Simple Java Ctr: without Lock

```
  @Override
  public void run() {
    while (counter < limit) {
      increaseCounter();
    }
  }
  private void increaseCounter() {
    System.out.println(Thread.currentThread().getName() + " : " + counter);
    counter++;
  }
}
```

1: Simple Java Ctr: with Sync Lock

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Counter implements Runnable {
    private static int counter = 0;
    private static final int limit = 1000;
    private static final int threadPoolSize = 5;
    private static final Object lock = new Object();
   public static void main(String[] args) {
        ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
        for (int i = 0; i < threadPoolSize; i++) { ES.submit(new Counter());   }
        ES.shutdown();
    }
}
```

1: Simple Java Ctr: with Sync Lock

```
    @Override
    public void run() {
        while (counter < limit) {
            increaseCounter();
        }
    }
    private void increaseCounter() {
        synchronized (lock) {
        System.out.println(Thread.currentThread().getName() + " : " + counter);
        counter++;
        }
    }
}
```

2: Simple Java Ctr: with atomic Int

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;
class Counter implements Runnable {
   private static AtomicInteger counter;
   private static final int limit = 1000;
   private static final int threadPoolSize = 5;
   public static void main(String[] args) {
      ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
      for (int i = 0; i < threadPoolSize; i++) { ES.submit(new Counter());   }
      ES.shutdown();
   }
}
```

2: Simple Java Ctr: with atomic int

```
  @Override
  public void run() {
    while (counter < limit) {
      increaseCounter();
    }
  }
  private void increaseCounter() {
   System.out.println(Thread.currentThread().getName() + " : " + counter);
    counter.getAndIncrement();


  }
}
```

3: Simple Java Ctr: Using concurrent lock

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.ReentrantLock;

class Counter implements Runnable {
    private ReentrantLock lock;
    private static final int limit = 1000;
    private static final int threadPoolSize = 5;

    public static void main(String[] args) {
        ReentrantLock sharedLock = new ReentrantLock();
        ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
        for (int i = 0; i < threadPoolSize; i++) { ES.submit(new Counter(sharedLock));   }
        ES.shutdown();
    }
}
```

3: Simple Java Ctr: Using concurrent lock

```
   public Counter(ReentrantLock lock) {
      this.lock = lock;
   }
@Override
   public void run() {
      while (counter < limit) {
         increaseCounter();
      }
   }
   private void increaseCounter() {
            lock.lock();
                  try { counter++;  }
                   finally {  lock.unlock(); }
}
```

4: Simple Java Ctr: Using a Semaphore

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
class Counter implements Runnable {
   Semaphore semaphore;
   private static final int limit = 1000;
   private static final int threadPoolSize = 5;
   public static void main(String[] args) {
        Semaphore sharedSemaphore = new Semaphore(1);
       ExecutorService ES = Executors.newFixedThreadPool(threadPoolSize);
       for (int i = 0; i < threadPoolSize; i++) {
                                        ES.submit(new Counter(sharedSemaphore));   }
       ES.shutdown();
   }
}
```

4: Simple Java Ctr: Using a Semaphore

```
public Counter(Semaphore semaphore){
            this.semaphore = semaphore;
  }
@Override
   public void run() {
      while (counter < limit) {
         increaseCounter();
      }
   }
   private void increaseCounter() {
         try {   semaphore.acquire(); counter++;
          } catch (InterruptedException e) {  e.printStackTrace();
               }  finally { semaphore.release();}
}
```

Basic Locking implementation in Java

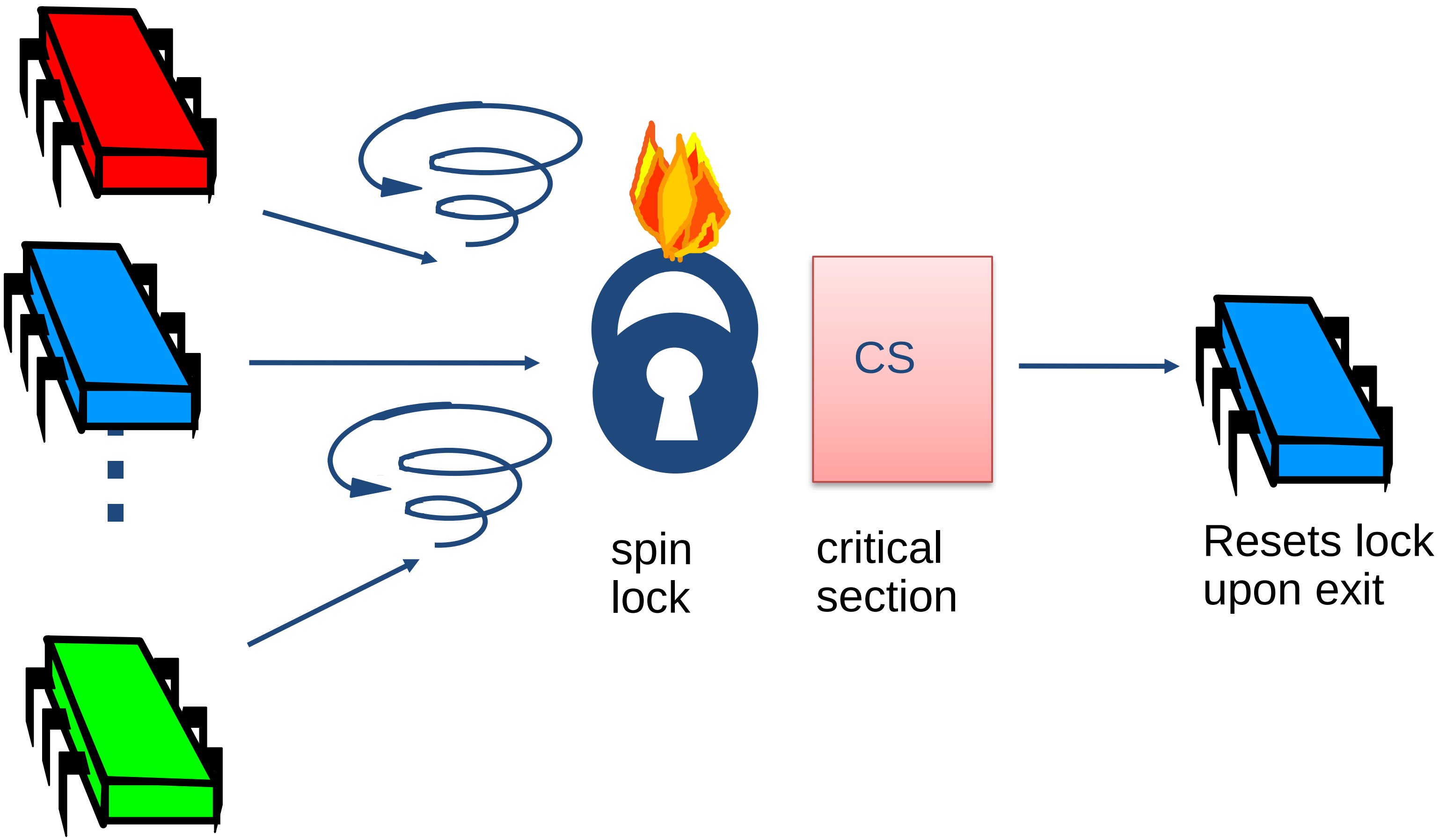
Basic Locking implementation in Java

- Locking require BASIC, Atomic Instruction at OS level
- CAS : Compare and Set /Test and Increment
- TAS : Test and Set
- TTAS: Try, Test and Set
- Exponential Back up TAS

Synchronization Hierarchy ☺ ☺ ☺

- One == (used by)== > other
- **LL+SC** ==> **TAS/CAS/FAI/XCHG**==>**Lock/Unlock**
  - All **TAS/CAS/GAS/FAI/XCHG** do the same work
- Lock/Unlock == > Mutex //Mutex use L/UL
- Mutex == > Semaphore // Semaphore uses Mutex
  - Wait() and Signal()
- Semaphore == > Monitor //Monitor uses Semaphore
  - Many wait/Many Signal, Processes in Queue
  - Monitor : Another Abstract Type
    - *which use semaphore, mutex, conditions*

Many threads trying to acquire Mutex LOCK : Spin Lock



CAS: CompareAndSet

```
public abstract class Lock{
 private int value;
 public boolean synchronized
   CompareAndSet(int expected,
                 int update) {
  int prior = value;
  if (value==expected) {
   value = update; return true;
  }
  return false;
  }
}
```

Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- TAS aka "getAndSet"

Test-and-Set

```
public class AtomicBoolean {
 boolean value;

 public synchronized boolean
  getAndSet(boolean newValue) {
   boolean prior = value;
   value = newValue;
   return prior;
 }
}
```

Test-and-set Lock : TASLock

```
class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {} //Spining
 }

 void unlock() {
  state.set(false);
 }
}
```

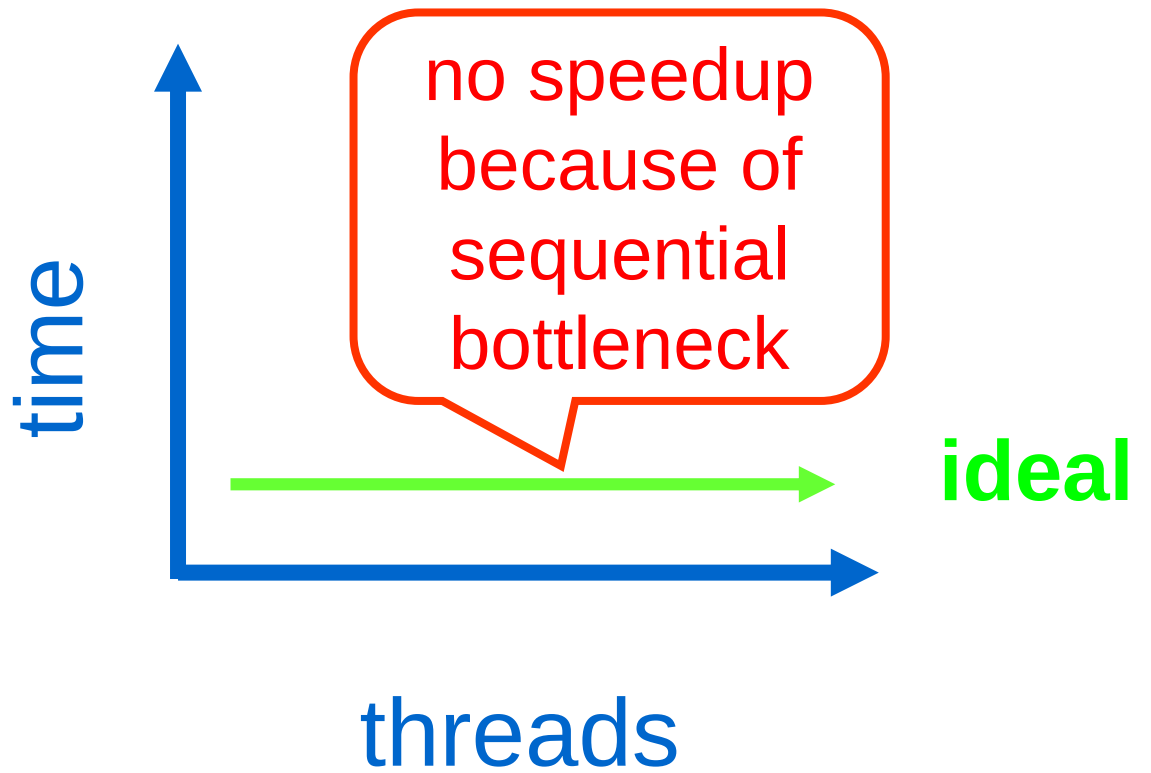
Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

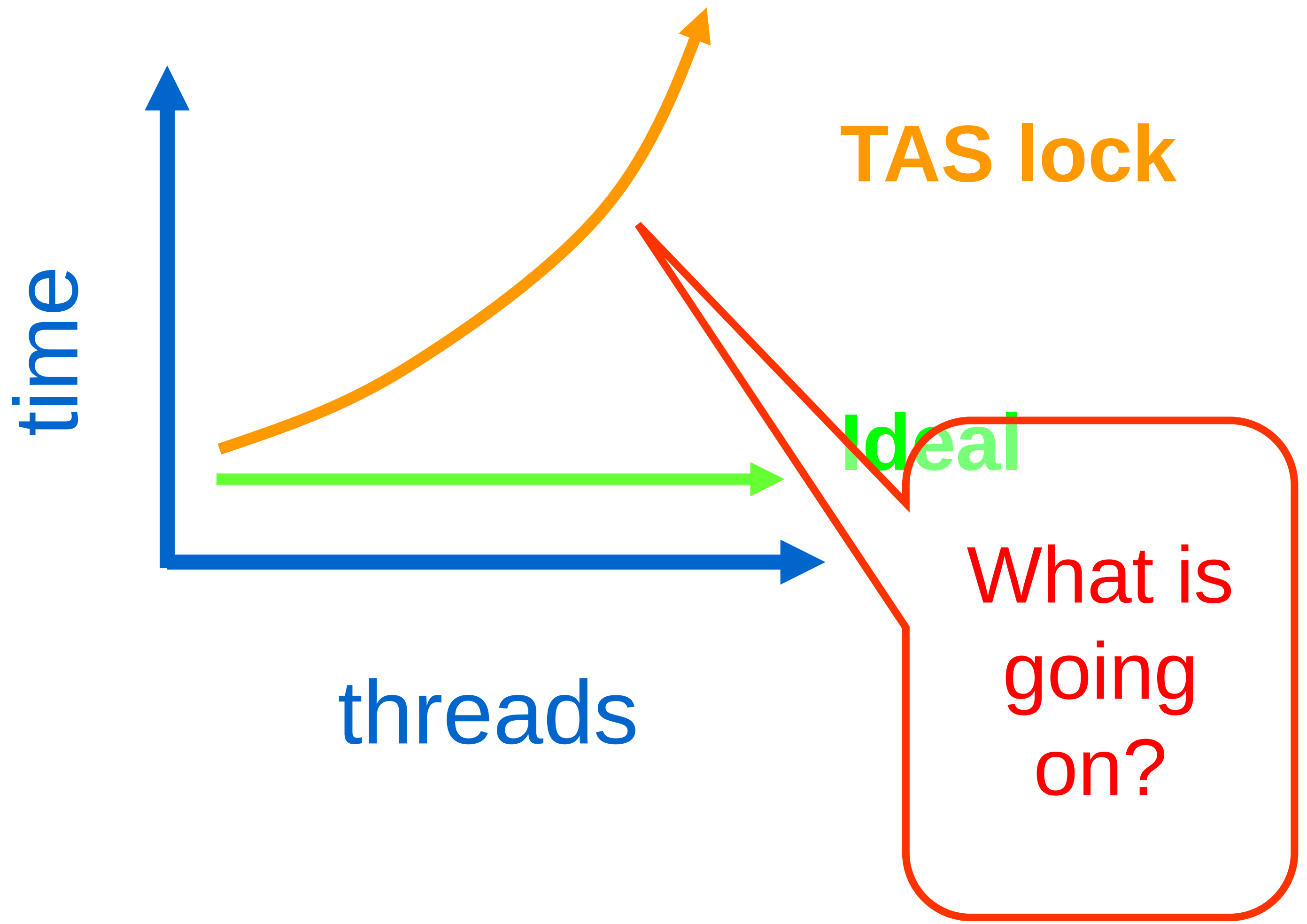
Performance

- Experiment
  - *n* threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

Graph


no speedup because of sequential bottleneck
ideal
time
threads

Mystery #1



Test-and-Test-and-Set Locks

- Lurking stage
  - Wait until lock “looks” free
  - Spin while read returns true (lock taken)
- Pouncing state
  - As soon as lock “looks” available
  - Read returns false (lock free)
  - Call TAS to acquire lock
  - If TAS loses, back to lurking

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

Wait until lock looks free

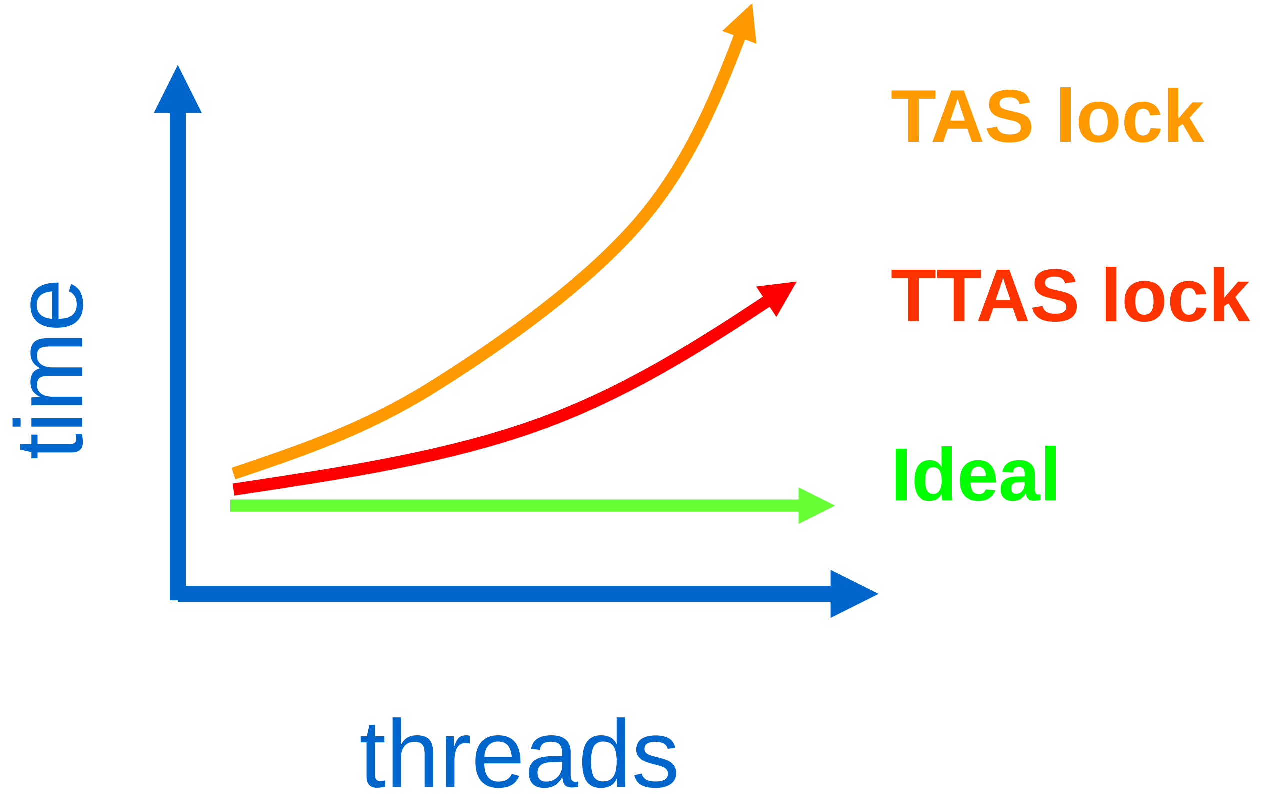
Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

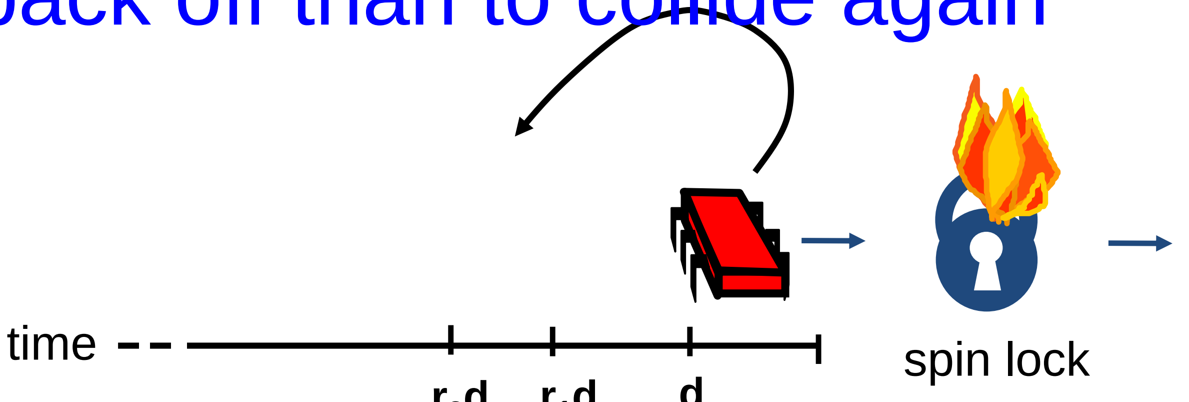
Then try to acquire it

Mystery #2

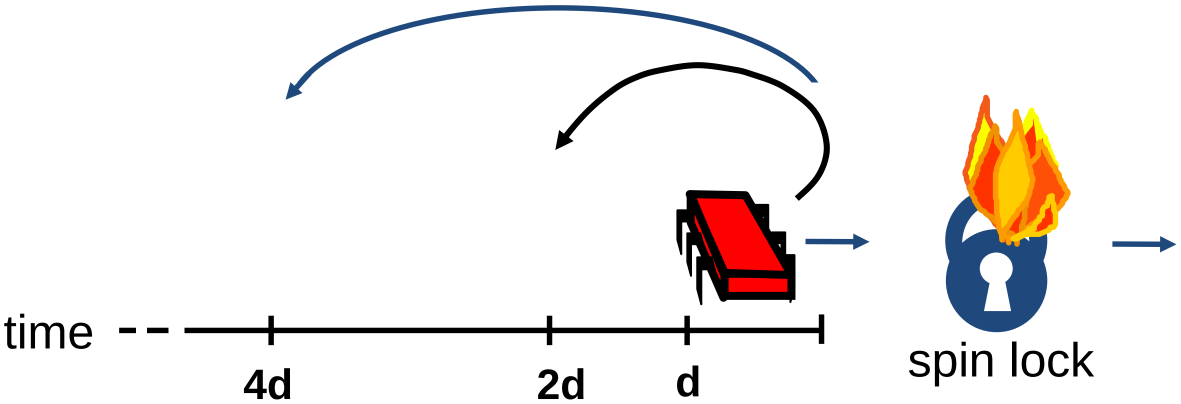


Solution: Introduce Delay

- If the lock looks free
  - But I fail to get it
- There must be contention
  - Better to back off than to collide again



Dynamic Example: Exponential Backoff



If I fail to get lock

- wait random duration before retry
- Each subsequent failure doubles expected wait

Exponential Backoff Lock

```
public class Backoff implements lock {
 public void lock() {
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

Exponential Backoff Lock

```
public class Backoff implements lock {
 public void lock() {
  int delay = MIN_DELAY;
   while (true) {
    while (state.get()) {}
    if (!lock.getAndSet(true))
     return;
    sleep(random() % delay);
    if (delay < MAX_DELAY)
     delay = 2 * delay;
 }}}
```

Fix minimum delay

Exponential Backoff Lock

```
public class Backoff implements lock {
 public void lock() {
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2
}}}
```

Wait until lock looks free

Exponential Backoff Lock

```
public class Backoff implements lock {
 public void lock() {
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

If we win, return

Exponential Backoff Lock

Back off for random duration

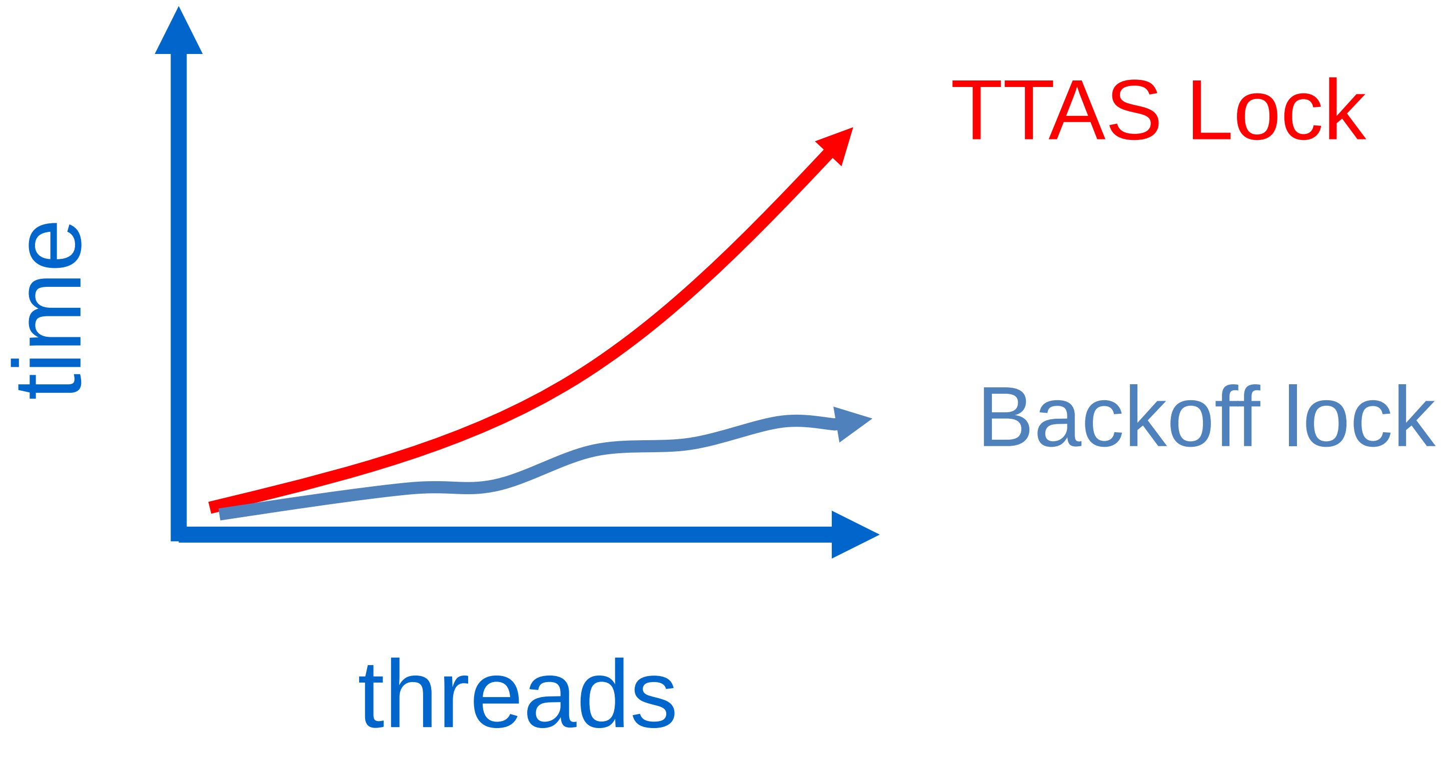
```
public
 publi
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

Exponential Backoff Lock

```
publ
 pub
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

Double max delay, within reason

Spin-Waiting Overhead



Backoff: Other Issues

- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully
  - Not portable across platforms

Concurrent/Thread-Safe Data Structure

CDS or TS-DS

Data Structure as Object

- Data Stored on DS
  - List, Queue, Stack , Hash, Tree
- Concurrent access should be safe
  - All operation including read, write, modify, add, remove
- Safe : It should not alter the properties of the DS

Today: Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks

Today: Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks
- Should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using queue locks
  - Easy to reason about
    - In simple cases
- So, are we done?

Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel …

Example : Linked List of Student

- Linked List of Students, with Roll No, Marks
- Roll No is Key
- Mark is Data

```
class node {
        int Roll;
        int Marks
       node next;
};
```

- Many thread simultaneously working on the LinkedList Object
- Operation to be performed
  - Read mark of a student
  - **Update mark of a student**
  - **Add a student**
  - **Delete a student**
  - Calculate average

Design of Concurrent DS

- Introduce four “patterns”
  - Bag of tricks …
  - Methods that work more than once …
- For highly-concurrent objects
  - Concurrent access
  - More threads, more throughput

First: Fine-Grained Synchronization

- Instead of using a single lock …
- Split object into
  - Independently-synchronized components
  - **Example: Hash : Modification**
- Methods conflict when they access
  - The same component …
  - At the same time

Second: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  – OK: we are done
  – Oops: start over
- Evaluation
  – Usually cheaper than locking, but
  – Mistakes are expensive

Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done

Fourth: Lock-Free Synchronization

- Don’t use locks at all
  - Use compareAndSet() & relatives …

Fourth: Lock-Free Synchronization

- Don’t use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

Thanks