CS331: Adv. Java Threads

http://jatinga.iitg.ac.in/~asahu/cs331/

A Sahu
Dept of Computer Science & Engineering
IIT Guwahati

Outline

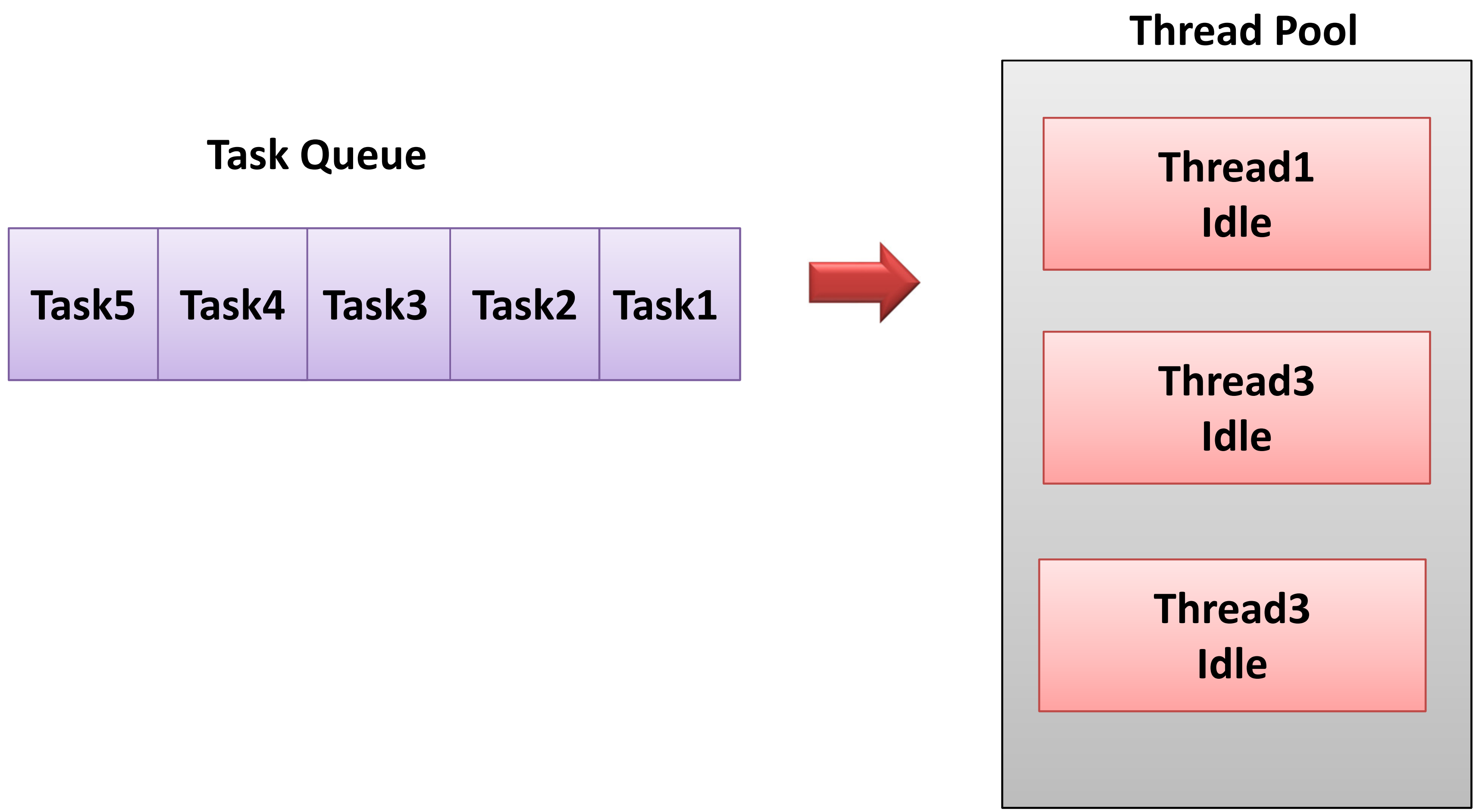
- Thread Pooling
- Thread Priority
- Java Collections: Data Structure
- Thread Safety: Sync, volatile, final, atomic
- Reentrant Lock
- Locking overhead: try lock, expo lock
- Fine grain Lock
- **Source Code Available@**

  http://jatinga.iitg.ac.in/~asahu/cs331/

Thread Pool in Java

- Thread pool reuses previously created threads
  - to execute current tasks
  - offers a solution to the problem of thread cycle overhead and resource thrashing
  - making the application more responsive
- We first create a object of **ExecutorService** and pass a set of tasks to it.
- **ThreadPoolExecutor** class allows to set the core and maximum pool size

Thread Pool in Java



Thread Pool in Java: Example

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

 class Task implements Runnable   { // Task class to be executed (Step 1)
   private String name;
   public Task(String s)     { name = s;  }
   // Prints task name and sleeps for 1s  and process is repeated 5 times
   public void run()     {
      try  {
         for (int i = 0; i<=5; i++)   {
            if (i==0)   { System.out.println("Init of task - "+ name +" = " +);          }
            else    {     System.out.println("Executing task "+       name);     }
            Thread.sleep(1000);
         }
         System.out.println(name+" complete");
      }
      catch(InterruptedException e)   {   e.printStackTrace();   }
   }
}
```

Thread Pool in Java: Example

```
public class ThreadPoolTest  {
    // Maximum number of threads in thread pool
    static final int MAX_T = 3;
    public static void main(String[] args)    {
      // creates six tasks //Step1
      Runnable r1 = new Task("task 1");    Runnable r2 = new Task("task 2");
      Runnable r3 = new Task("task 3");    Runnable r4 = new Task("task 4");
      Runnable r5 = new Task("task 5");    Runnable r6 = new Task("task 5");
       // creates Thread pool with MAX_T no. of   threads //Step2
      ExecutorService pool = Executors.newFixedThreadPool(MAX_T);
      // passes the Task objects to the pool to execute (Step 3)
      pool.execute(r1);     pool.execute(r2);    pool.execute(r3);
      pool.execute(r4);      pool.execute(r5);   pool.execute(r6);
      // pool shutdown ( Step 4)
      pool.shutdown();
   }
}
```

Java thread priority

- Java Thread Priority: ranging from 1 to 10
- Value:
  - NORM_PRIORITY=5; default
  - MIN_PRIORITY=1;
  - MAX_PRIORITY=10
- Methods to change priority
  - **int getPriority()**
  - **setPriority(int newPriority); //1-10**

Java thread priority

```
public class PriThrd extends Thread{
static int x = 0;
String name; PriThrd (String n) { name = n; }
public void increment() {
        x = x+1; System.out.println(x + " " + name);
        }
public void run() {  while(1) this.increment();  }
 }
public class PriThrdTest {
  public static void main(String args[]) {
        PriThrd a = new PriThrd1("a");
        PriThrd b = new PriThrd2("b");
        a.setPriority(10); b.setPriority(1);
        a.start(); b.start();
        }
}
```

Java Data Structure Collections

- Vector
- Stack
- Queue/Deque
  - Double ended queue: ins/del from both end
- Priority queue
- Linedlist
- HashSet: Hash Table
  - Set
- HashMap:
  - for Map, array with non-int indexing

Synchronized, Volatile, Atomic

```
class CounterThread extends Thread {
        private int count;
// Synchronized method to prevent race conditions
@Override
 public synchronized void run() {
        count++; // Increment the count
  }
 public int getCount() { return count; }
}
```

Synchronized, Volatile, Atomic

- **Working of Synchronized Modifier:**
  - It can be applied to methods or blocks of code.
  - When a method or block is synchronized, only one thread can execute it on a given object at any time.
  - Every object in Java has an **intrinsic lock** associated with it. A thread must acquire this lock before entering a synchronized method or block.
  - Synchronized code blocks may lead to thread contention, which can negatively impact performance, especially with excessive synchronization.

Synchronized, Volatile, Atomic

```
class Counter {
 private volatile int count; // Volatile variable
 public void increment() {
        count++; // This operation is not atomic
   }
  public int getCount() { return count; }
}
```

Synchronized, Volatile, Atomic

- The volatile keyword in Java ensures that
  - all threads have a consistent view of a variable’s value.
  - It prevents caching of the variable’s value by threads, ensuring that updates to the variable are immediately visible to other threads.
- **Working of Volatile Modifier:**
  - It applies only to variables.
  - **volatile** guarantees visibility i.e. any write to a volatile variable is immediately visible to other threads.
  - It does not guarantee atomicity, meaning operations like count++ (read-modify-write operations) can still result in inconsistent values.

Synchronized, Volatile, Atomic

```
import java.util.concurrent.atomic.AtomicInteger;
class CounterThread extends Thread {
 private AtomicInteger count = new AtomicInteger();
// Atomic variable @Override
public void run() {
 count.incrementAndGet(); // Atomic increment
 }
 public int getCount() { return count.get();}
 }
```

Synchronized, Volatile, Atomic

- **Atomic classes**, such as **AtomicInteger**,
  - are part of the **java.util.concurrent.atomic** package.
  - provide thread-safe operations on variables without the need for synchronization.
  - They use low-level atomic operations like compare-and-swap (CAS)/TAS to ensure thread safety.
- **Working of Atomic Modifier:**
  - Atomic operations ensure atomicity of the read-modify-write actions on variables.
  - These classes are lock-free and more efficient than synchronized blocks because they avoid the overhead of acquiring locks.
  - Atomic operations are performed using methods like **incrementAndGet(), compareAndSet(), and getAndSet()**.

Re-entrant Lock

- Achieve synch more effectively and optimally
  - offers features like timeouts, interruptible locks, and more control over thread scheduling
- Allows a thread to acquire the same lock
  - multiple times (**without blocking**), which is particularly useful
  - when a thread needs to access a shared resource repeatedly within its execution
- **Re-entrantLock tracks a "hold count"**
  - which is a value that starts at 1 when a thread first locks the resource.
  - Each time the thread re-enters the lock, the count is incremented.
  - The count is decremented when the lock is released.
  - Once the hold count reaches zero, the lock is fully released.

Re-entrant Lock: Example

```
class ReLockExample {
        private static int c = 0;
        static ReentrantLock lock = new ReentrantLock();
public static void increment() {
        // acquire the lock
        lock.lock();     try { c++; }
        finally { lock.unlock(); } //release the lock
}
public static void main(String[] args) {
        Runnable task = () -> { for (int i = 1; i < 3 ; i++) { increment(); }
                };
        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");
        t1.start(); t2.start();
        }
}
```

Java: notify(); wait();

```
class Buffer {
    private final int[] buffer;      private final int size;     private int count;
    public Buffer(int size) {
        this.size = size;      this.buffer = new int[size];      this.count = 0;
    }
    public synchronized void produce(int item) throws InterruptedException {
        while (count == size) {wait();   } // Buffer is full, wait for the consumer to consume
        buffer[count] = item;      count++;      System.out.println("Produced: " + item);
        notify(); // Notify the consumer that an item is available
    }

   public synchronized int consume() throws InterruptedException {
        while (count == 0) {    wait();    } // Buffer is empty, wait for the producer to produce
        int item = buffer[count - 1];    count--;
        System.out.println("Consumed: " + item);
        notify(); // Notify the producer that space is available
        return item;
    }
}
```

Java: notifyall(); wait();

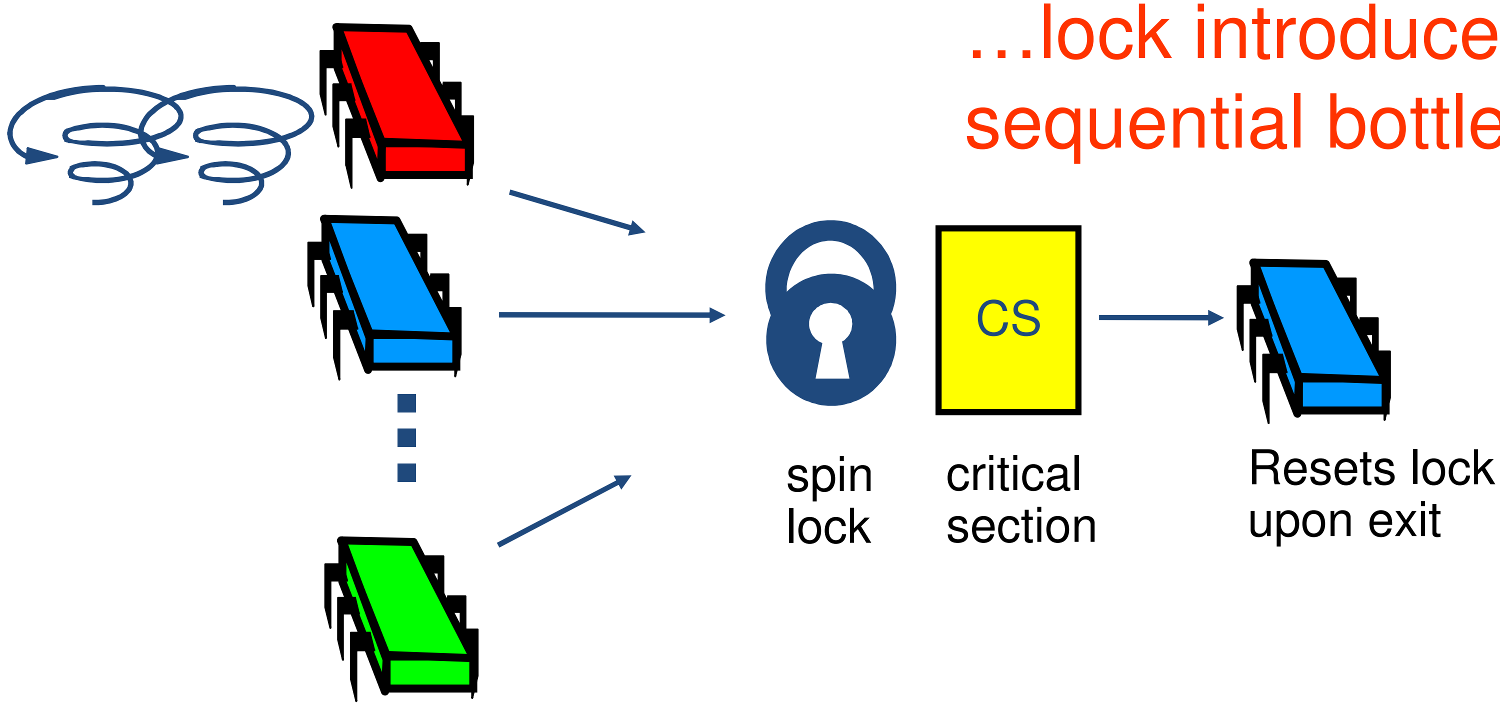
```
class ChatRoom {
    private Queue<String> messages = new LinkedList<>();
    public synchronized void sendMessage(String message) {
        messages.add(message);
        System.out.println("Message sent: " + message);
        notifyAll(); // Notify all users about the new message
    }
    public synchronized String receiveMessage() throws InterruptedException {
        while (messages.isEmpty()) {
            wait(); // Wait if no messages are available
        }
        String message = messages.poll();
        System.out.println("Message received: " + message);
        return message;
    }
}
```

Java: notifyall(); wait();

```
class User implements Runnable {
    private String name;
    private ChatRoom chatRoom;
    public User(String name, ChatRoom chatRoom) {
        this.name = name;     this.chatRoom = chatRoom;
    }
    @Override
    public void run() {
        try {     for (int i = 1; i <= 5; i++) {
                String message = "Hello from " + name + " - Message " + i;
                chatRoom.sendMessage(message);
                Thread.sleep(100); // Simulate some work before sending the next message
            }
        } catch (InterruptedException e) {  Thread.currentThread().interrupt();  }
    }
}
```

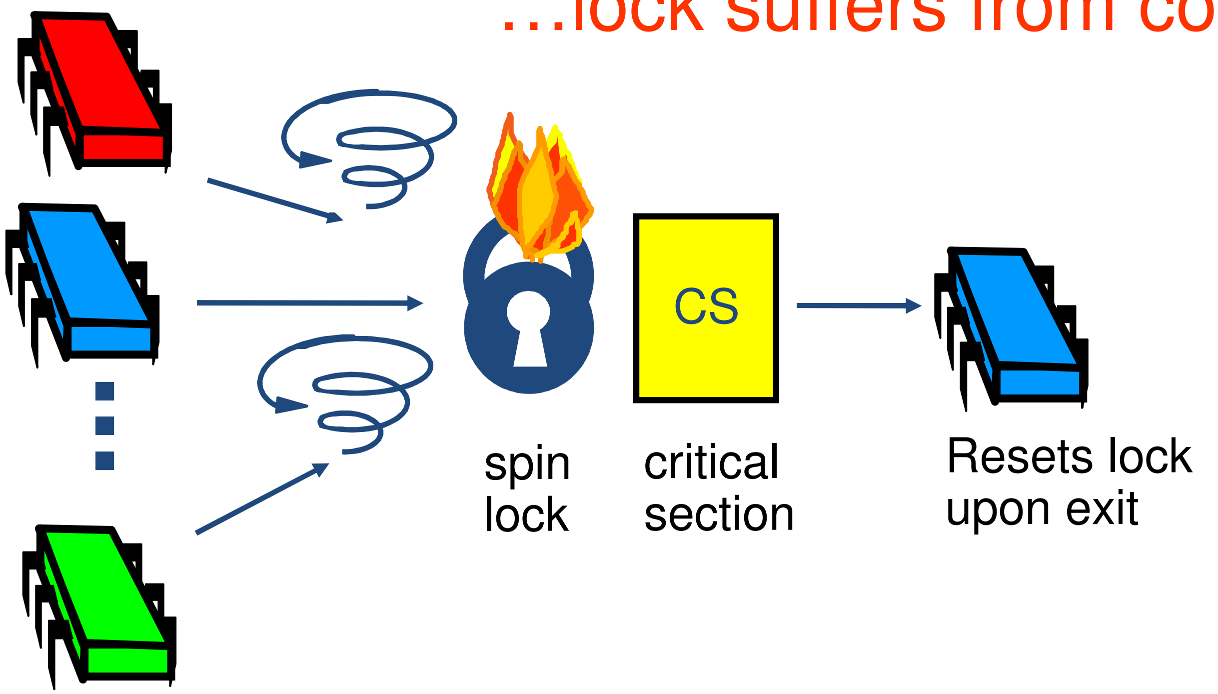
Basic Spin-Lock

…lock introduces sequential bottleneck



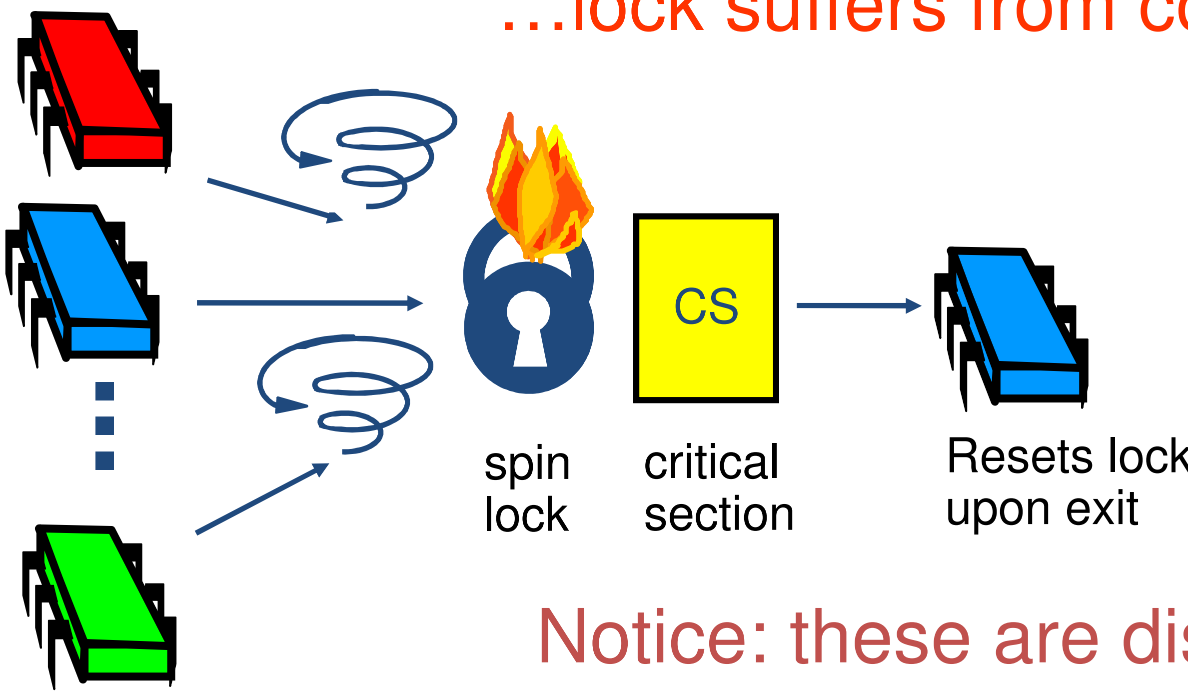
Basic Spin-Lock

…lock suffers from contention



Basic Spin-Lock

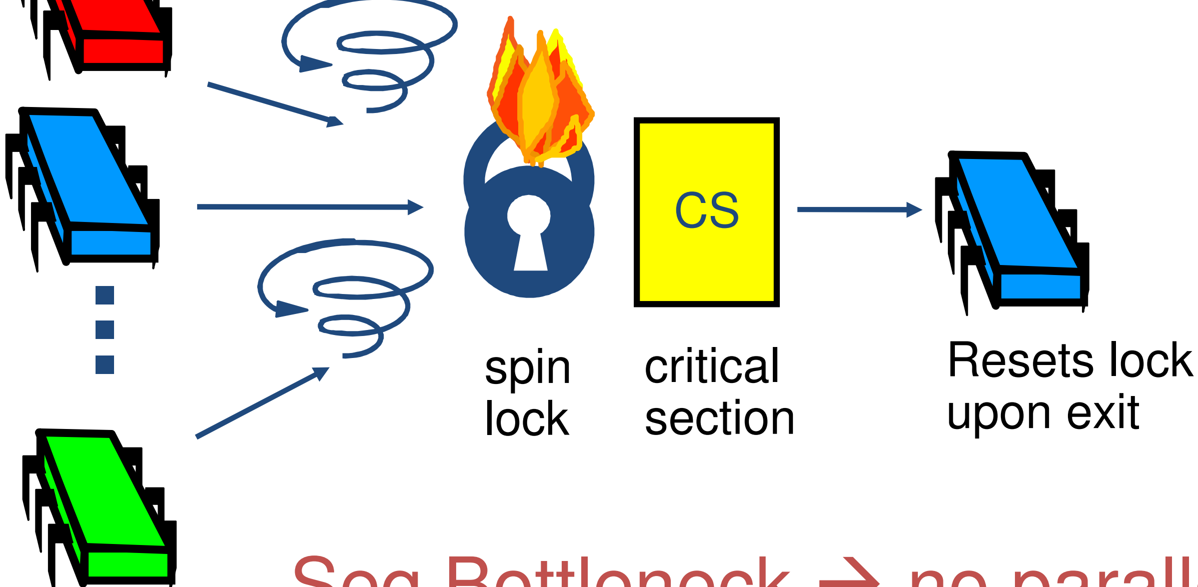
...lock suffers from contention



Notice: these are distinct phenomena

Basic Spin-Lock

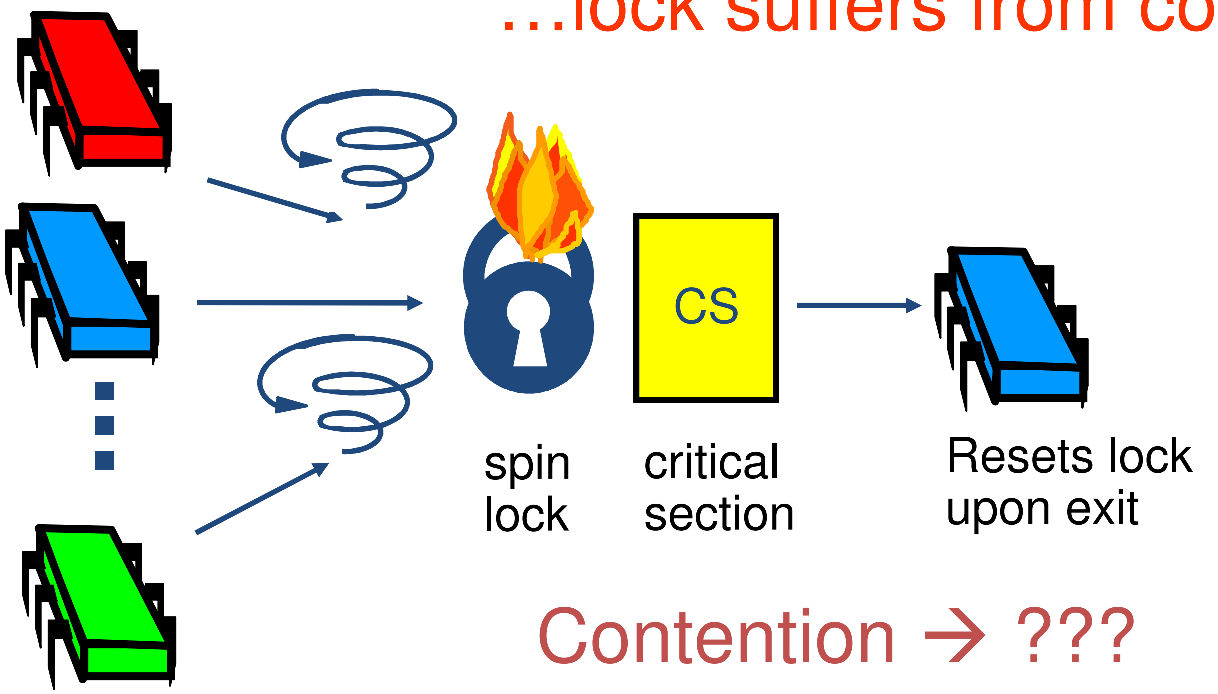
...lock suffers from contention



Seq Bottleneck → no parallelism

Basic Spin-Lock

...lock suffers from contention



Contention → ???

Review: 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”

Review: Test-and-Set

import java.util.concurrent.atomic

```
public class AtomicBoolean {
 boolean value;

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

Swap old and new values

Test-and-Set Locks

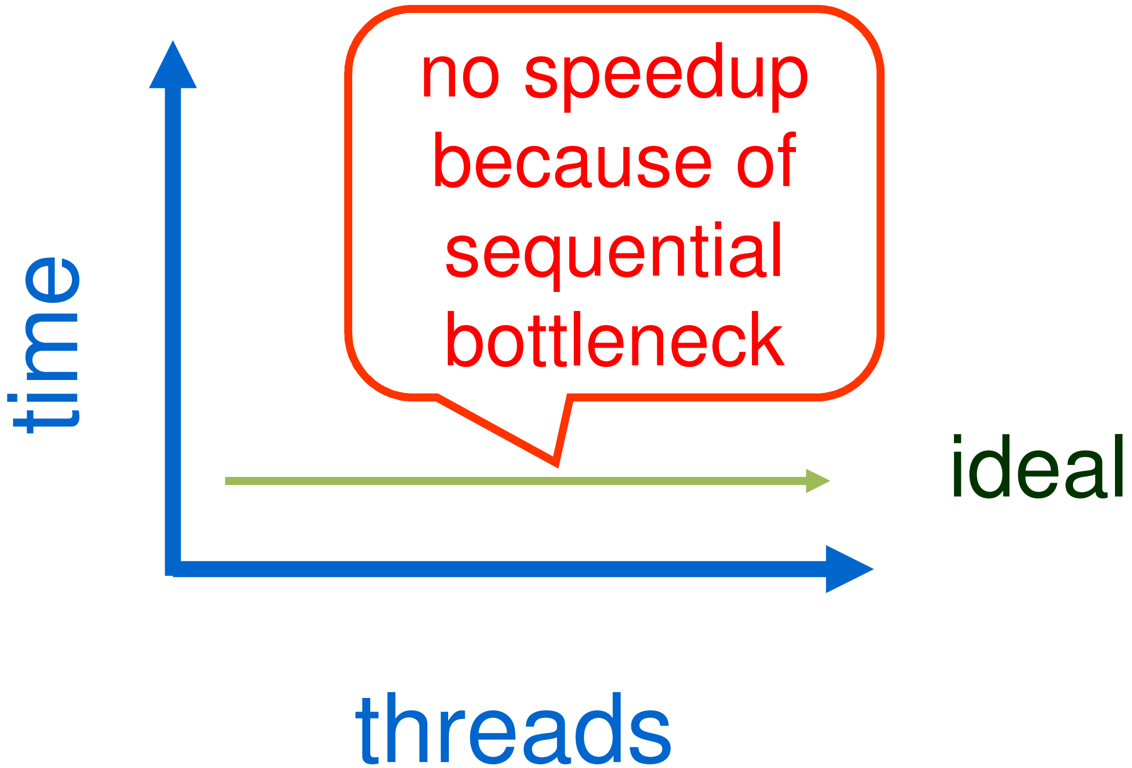
- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

Test-and-set Lock

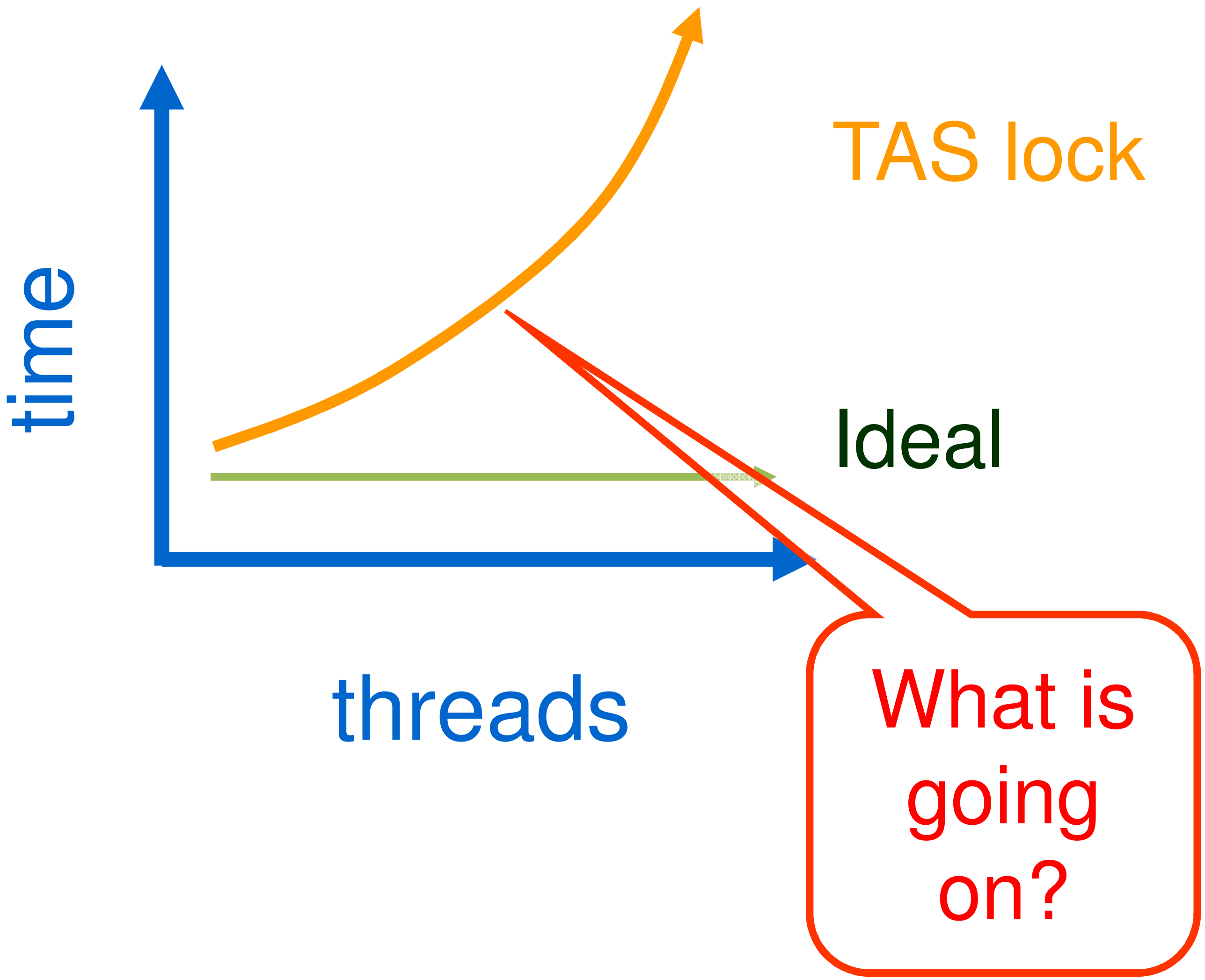
```
class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

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

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

Keep trying until lock acquired

Graph


no speedup
because of
sequential
bottleneck
time
ideal
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

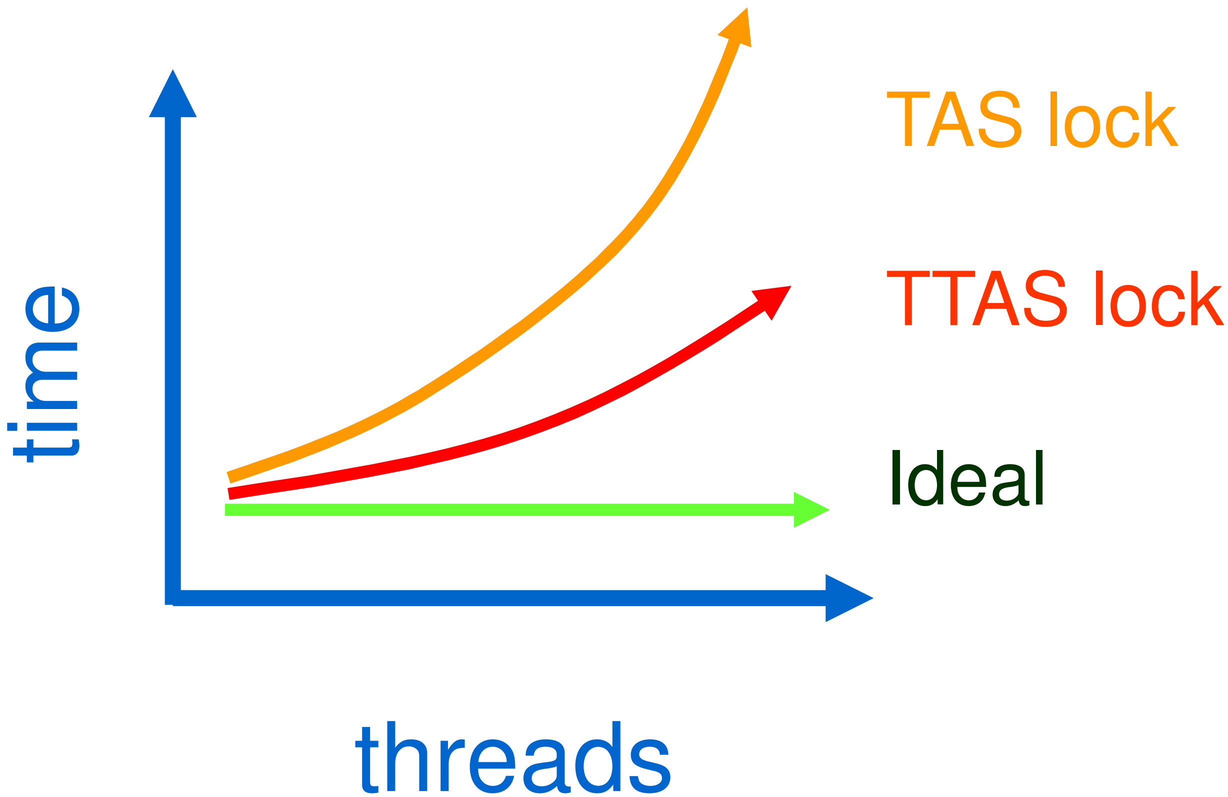
Test-and-test-and-set Lock

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

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

Then try to acquire it

Wait until lock looks free

Mystery #2



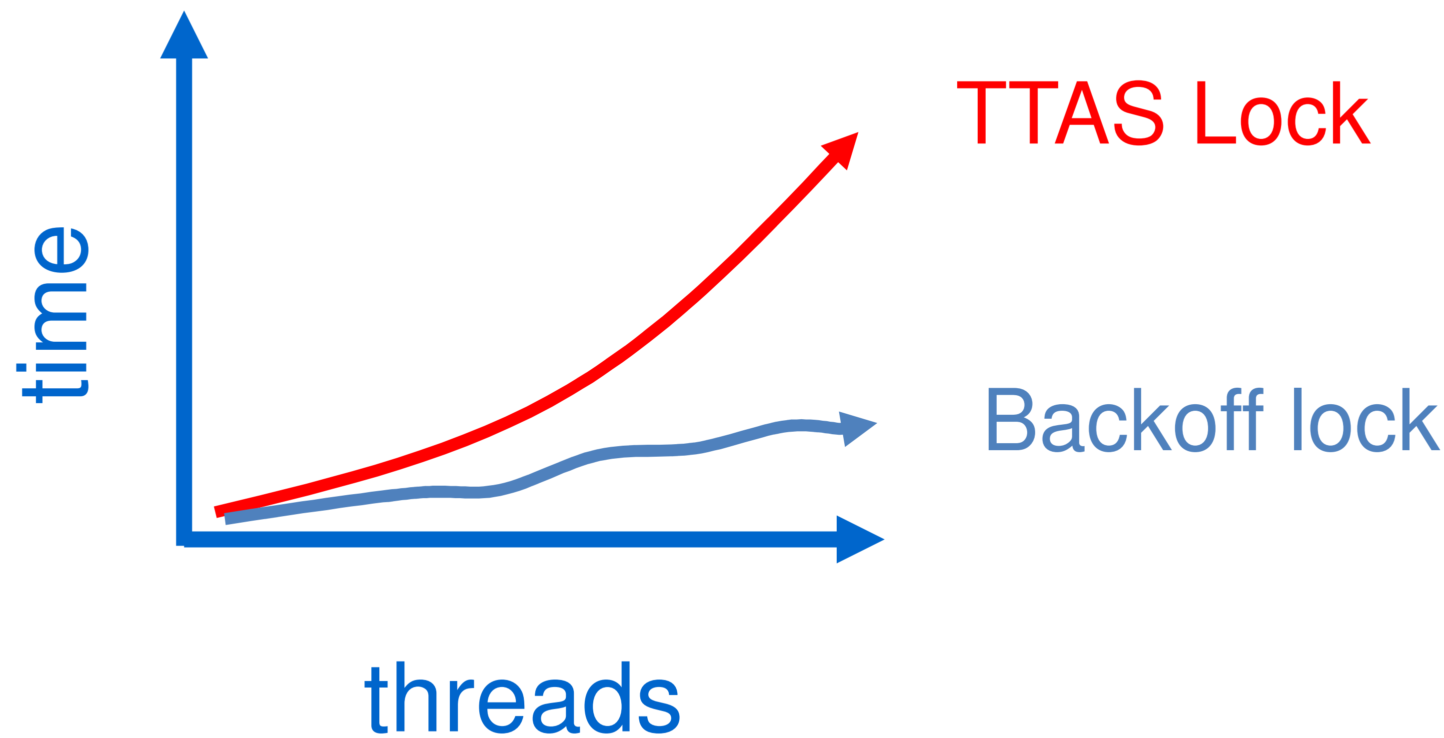
Mystery

- Both
  - TAS and TTAS
  - Do the same thing (in our model)
- Except that
  - TTAS performs much better than TAS
  - Neither approaches ideal
- Approach : Similar to CSMA BUS protocol
  - If many people are waiting for shared lock/Lock is busy.. Let me wait for some time then try
  - Waiting time may be fixed or increased exponentially.

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;
    }
 }
}
```

Spin-Waiting Overhead



Fine Grain Lock Vs Coarse Grain Lock

Coarse Lock

```
class CoarseLock {
        vector <ACC>  A;  //vector hold balance of acc
    static ReentrantLock lock = new ReentrantLock();
public static void ModifyVal(int D, int V) {
    // acquire the lock
    lock.lock();      try { A[D] +=V; }
    finally { lock.unlock(); } //release the lock
}
public static void AddAcc( ACC a){      //Fine for this Operation
    lock.lock();      try {A.add(a);}
    finally { lock.unlock(); } //release the lock
 }
}
```

Finally used after try-catch block finished: try{} catch {} finally{}; finally block always execute

Fine grain Lock

```
class CoarseLock {
vector <ACC>  A;  //vector hold balance of acc
        static ReentrantLock lock = new ReentrantLock();
public static void ModifyVal(int D, int V) {
        A[D].addBal(V);
   }
}
Class ACC{
     private static bal=0;
     public synchronized  addBal(int V){ bal+=V; } //sync for  only this ACC obj
}
```

Lock free DSA

- Lock-free algorithms provide a way
  - in which threads can access the shared resources
  - without the complexity of Locks and
  - **without blocking the threads forever**
- Lock free DS: become a programmer’s choice
  - as they **provide higher throughput** and
  - prevent deadlocks.

Classic Stack for Concurrency

```
private static class ClassicStack<T> {
        private StackNode<T> headNode;
        private int noOfOps;
        // Synchronizing the operations  for concurrency control
        public synchronized int getNoOfOps()   {  return noOfOps;  }
        public synchronized void push(T number)   {
          StackNode<T> newNode = new StackNode<T>(number);
          newNode.next = headNode;
          headNode = newNode;   noOfOpss++;
        }
```

Classic Stack for Concurrency

```
public synchronized T pop()   {
        if (headNode == null)     return null;
        else {   T val = headNode.getValue();
            StackNode<T> newHead = headNode.next;
            headNode.next = newHead;   noOfOperations++;
            return val;
        }
    }
private static class StackNode<T> {
    T value;
    StackNode<T> next;
    StackNode(T value) { this.value = value; }
      public T getValue() { return this.value; }
  }
}
```

Lock free Stack for Concurrency

```
private static class LockFreeStack<T> {
     // Defining the stack nodes as Atomic Reference
     private AtomicReference<StackNode<T> > headNode
       = new AtomicReference<StackNode<T> >();
     private AtomicInteger noOfOperations
       = new AtomicInteger(0);
      public int getNoOfOperations()    {
       return noOfOperations.get();
     }
```

Lock free Stack for Concurrency

```
public void push(T value)    {
        StackNode<T> newHead = new StackNode<T>(value);
         // CAS loop defined
        while (true) {
          StackNode<T> currHeadNode     = headNode.get();
          newHead.next = currHeadNode;
           // perform CAS operation before setting new  value
          if (headNode.compareAndSet(curHeadNode,  newHead)) break;
          else {  LockSupport.parkNanos(1);  }  // waiting for a nanosecond
        }
         noOfOperations.incrementAndGet(); // getting the value atomically
      }
```

Lock free Stack for Concurrency

```
public T pop()     {
        StackNode<T> curHeadNode = headNode.get();
         // CAS loop defined
        while (curHeadNode != null) {
           StackNode<T> newHead = curHeadNode.next;
           if (headNode.compareAndSet(curHeadNode, newHead))    break;
           else {                    // waiting for a nanosecond
              LockSupport.parkNanos(1);
              curHeadNode = headNode.get();
           }
        }
        noOfOperations.incrementAndGet();
        return curHeadNode != null  ? curHeadNode.value  : null;
     }
```