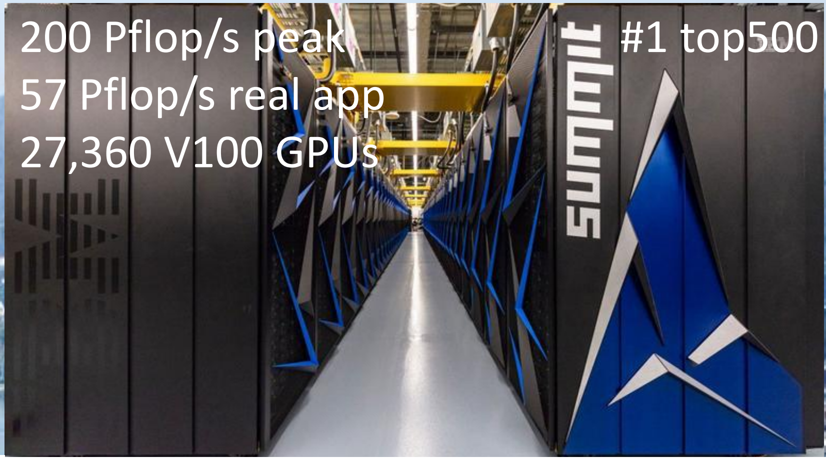
ETH zürich

spcl.inf.ethz.ch
@spcl_eth

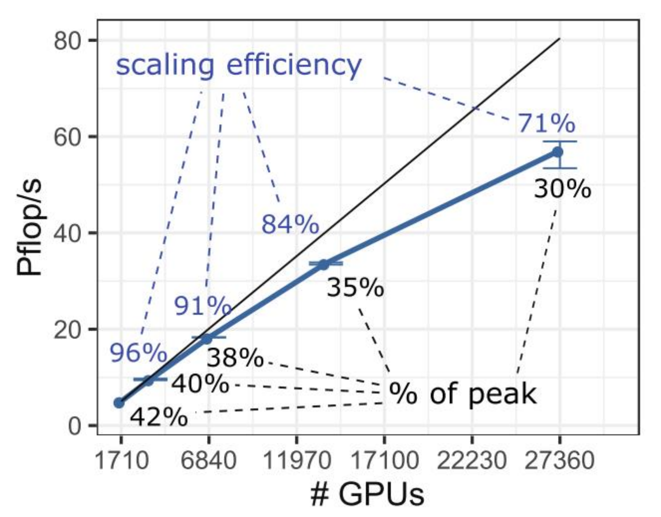
D INFK

TORSTEN HOEFLER

Parallel Programming, Spring 2019, Lecture 16+1:
Spinlocks, Deadlocks, Semaphores



What we did this weekend ☺
(the weather was bad anyway)



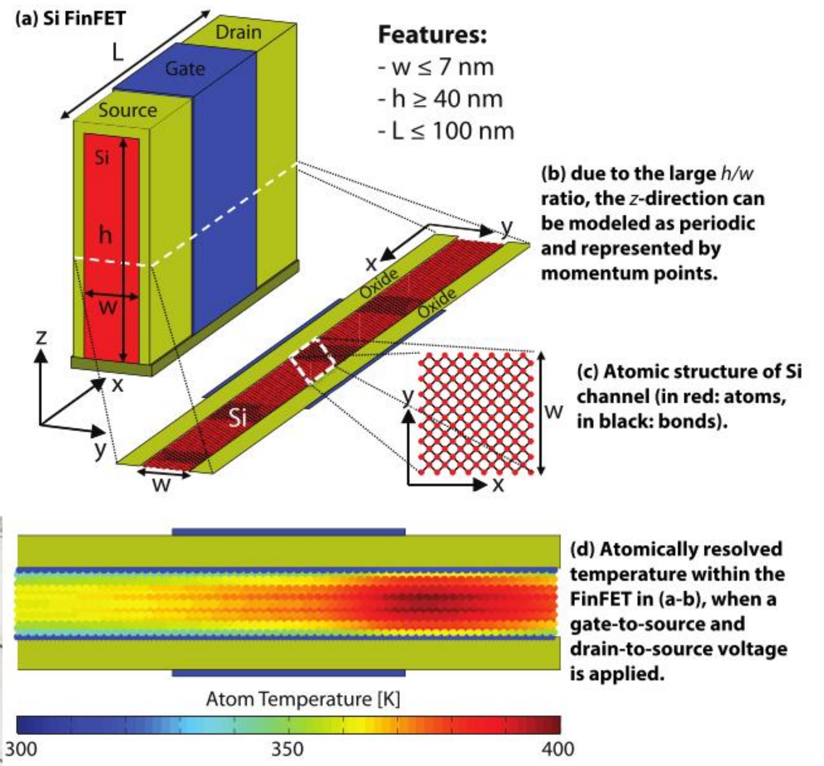


Figure 1: Simulation of self-heating effects in a 3-D Silicon FinFET with the OMEN code.

Outline

- **So far:**
  - Bad interleavings and data races and why they happen
  - Memory ordering and how we formalize it to drive proofs
  - Implementation of locks using atomic or safe registers (Peterson + Filter lock)

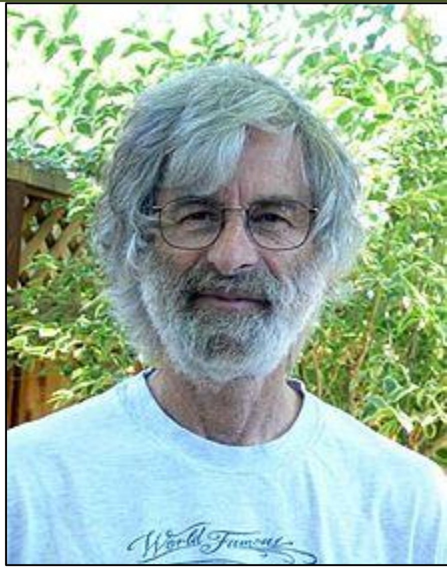
- **Now:**
  - Multi-process locks (using SWMR registers)
  - Implementation of locks with Read-Modify-Write operations
  - Concurrency on a higher level: Deadlocks, Semaphores, Barriers

- **Learning goals:**
  - Understand pitfalls in very simple synchronization algorithms
  - This is very important to design correct parallel codes

Mutual Exclusion for n processes: Bakery Algorithm (1974)

A process is required to take a numbered ticket with value greater than all outstanding tickets

CS Entry: Wait until ticket number is lowest



Lamport, Turing award 2013

Bakery algorithm (two processes, simplified)

```
volatile int np = 0, nq = 0
```

Process P

```
loop
    non-critical section
    np = nq + 1
    while (nq != 0 && nq < np);
    critical section
    np = 0
```

Q also wants access

and Q has an earlier ticket

Process Q

```
loop
    non-critical section
    nq = np + 1
    while (np != 0 && np <= nq)
    critical section
    nq = 0
```

np == nq can happen
➔ global ordering of processes

Bakery algorithm (n processes)

```
integer array[0..n-1] label = [0, ..., 0]
boolean array[0..n-1] flag =  [false, ..., false]
```

SWMR «ticket number»

SWMR «I want the lock»

```
lock(me):
   flag[me] = true;
   label[me] = max(label[0], ... , label[n-1]) + 1;
   while (∃k ≠ me: flag[k] && (k,label[k]) <l (me, label[me])) {};

unlock(me):
   flag[me] = false;
```

$(k, l_k) <_l (j, l_j) \Leftrightarrow l_k < l_j \text{ or } (l_k = l_j \text{ and } k < j)$

Bakery Lock in Java

Nice lock! But which problem remains?

```
class BakeryLock
{
   AtomicIntegerArray flag; // there is no
AtomicBooleanArray
   AtomicIntegerArray label;
   final int n;

   BakeryLock(int n) {
      this.n = n;
      flag = new AtomicIntegerArray(n);
      label = new AtomicIntegerArray(n);
   }

   int MaxLabel() {
      int max = label.get(0);
      for (int i = 1; i<n; ++i)
         max = Math.max(max, label.get(i));
      return max;
   }
   ...
```

```
   boolean Conflict(int me) {
      for (int i = 0; i < n; ++i)
         if (i != me && flag.get(i) != 0) {
            int diff = label.get(i) - label.get(me);
            if (diff < 0 || diff == 0 && i < me)
               return true;
         }
      return false;
   }

   public void Acquire(int me) {
      flag.set(me, 1);
      label.set(me, MaxLabel() + 1);
      while(Conflict(me));
   }

   public void Release(int me) {
      flag.set(me, 0);
   }
}
```

In general

Shared memory locations (atomic registers) come in different variants

- **Multi-Reader-Single-Writer (flag[], label[] in Bakery)**
- **Multi-Reader-Multi-Writer (victim in Peterson)**

- **Theorem 5.1 in [1]: *"If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes"***

INFORMATION AND COMPUTATION 107, 171–184 (1993)

[1]: Bounds on Shared Memory for Mutual Exclusion*

JAMES E. BURNS

Georgia Institute of Technology, Atlanta, Georgia 30332

AND

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

The shared memory requirements of Dijkstra's mutual exclusion problem are examined. It is shown that n binary shared variables are necessary and sufficient to solve the problem of mutual exclusion with guaranteed global progress for n processes using only atomic reads and writes of shared variables for communication.

We have constructed something …

… that may not quite fulfil its purpose!

AND we cannot do better, can we?

- **Is mutual exclusion really implemented like this?**
  - NO! Why?
    - *space lower bound linear in the number of maximum threads!*
    - *without precautions (volatile variables) our assumptions on memory reordering does not hold. Memory barriers in hardware are expensive.*
    - *algorithms are not wait-free (more later)*

- **But we proved that we cannot do better. What now!?**
  - Change (extend) the model with architecture engineering!
    - *modern multiprocessor architectures provide special instructions for atomically reading and writing at once!*

Hardware Support for Parallelism

Read-Modify-Write Operations

Hardware support for atomic operations: Example (x86)

CMPXCHG **Compare and Exchange**

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0.

The OF, SF, AF, PF, and CF flags are set to reflect the results of the compare.

When the first … modify-write on the memory operand … same value to the memory operand …

The forms of th… K prefix. For details about the LOCK …

Mnemonic

CMPXCHG reg… register or memory … operand to the first … operand to AL.

CMPXCHG reg… register or memory … operand to the first … operand to AX.

CMPXCHG reg… register or memory … operand to the first … operand to EAX.

CMPXCHG *reg/mem64*, *reg64* 0F B1 /r … register or mem… location. If equal, copy the second operand to the fi… operand. Otherwise, copy the first operand to RAX.

Related Instructions

CMPXCHG8B, CMPXCHG16B

CMPXCHG mem, reg
«compares the value in Register A with the value in a memory location If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the flag registers to 1. Otherwise it copies the value in the first operand to the A register and clears ZF flag to 0»

1.2.5 Lock Prefix

The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve

8 *Instruction Formats*

AMD

24594—Rev. 3.14—September 2007 *AMD64 Technology*

bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The LOCK prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if the LOCK prefix is used with any other instruction.

«The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically»

From the AMD64 Architecture Programmer's Manual

R. Hudson: IA memory ordering: https://www.youtube.com/watch?v=WUfvvFD5tAA (2008)

Hardware support for atomic operations: Example (ARM)

LDREX

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 1 0 0	1	Rn	Rd	SBO	1 0 0 1	SBO

LDREX (Load Register Exclusive) loads a register from memory, and:

- if the address has ... ess for the executing process...
- causes the execut...

Syntax

LDREX{<cond>} <Rd>, [<...

where:

<cond> Is the co... ...ned in *The conditio*... ...sed.

<Rd> Specifie...

<Rn> Specifies the register containing the address.

Architecture version

Version 6 and above.

LDREX <rd>, <rn>
«Loads a register from memory and if the address has the shared memory attribute, mark the physical address as exclusive access for the executing processor in a shared monitor»

STREX

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 1 0 0	0	Rn	Rd	SBO	1 0 0 1	Rm

STREX (Store Register Exclusive) performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

Syntax

STREX{<cond>} <R...

where:

<cond> I... ...e defined in *The* ...on is used.

<Rd> S... ...returned is:

0 if the operation updates memory

1 if the operation fails to update memory.

STREX <rd>, <rm>, <rn>
«performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed»

From the ARM Architecture Reference Manual

Hardware support for atomic operations

Atomic instructions are typically much slower than simple read & write operations [1]!

Typical instructions

Test-And-Set (TAS)

Example TSL register, flag (Motorola 68000)

Compare-And-Swap (CAS)

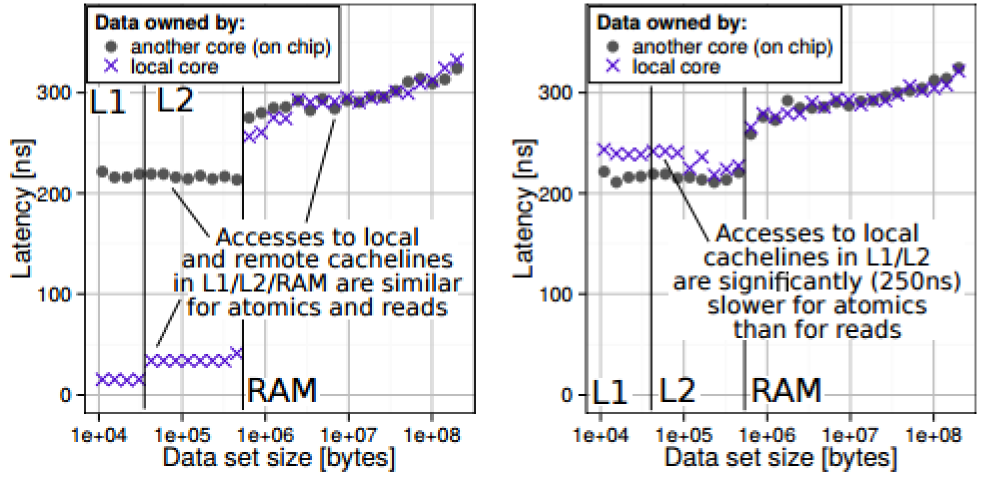
Example: LOCK CMPXCHG (Intel x86)

Example: CASA (Sparc)

Load Linked / Store Conditional

Example LDREX/STREX (ARM)

Example LL / SC (MIPS, POWER, RISC V)



(a) CAS, M/E state (b) CAS, S state

Fig. 6: The comparison of the latency of CAS on Xeon Phi. The requesting core accesses its own cache lines (local) and cache lines of a different core from the same chip (on chip).

[1]: H. Schweizer, M. Besta, T. Hoefler: Evaluating the Cost of Atomic Operations on Modern Architectures, ACM PACT'15

Semantics of TAS and CAS

```
boolean TAS(memref s)
  atomic
    if (mem[s] == 0) {
        mem[s] = 1;
        return true;
    } else
        return false;
```

```
int CAS (memref a, int old, int new)
  atomic
    oldval = mem[a];
    if (old == oldval)
        mem[a] = new;
    return oldval;
```

- are **Read-Modify-Write** («atomic») operations
- enable implementation of a mutex with O(1) space (in contrast to Filter lock, Bakery lock etc.)
- are needed for lock-free programming (later in this course)

Implementation of a spinlock using simple atomic operations

Test and Set (TAS)

```
Init (lock)
    lock = 0;
```

```
Acquire (lock)
    while !TAS(lock); // wait
```

```
Release (lock)
    lock = 0;
```

Compare and Swap (CAS)

```
Init (lock)
    lock = 0;
```

```
Acquire (lock)
    while (CAS(lock, 0, 1) != 0);
```

```
Release (lock)
    CAS(lock, 1, 0);
```

ignore result

Read-Modify-Write in Java

Let's try it.

Need support for atomic operations on a high level.

Available in Java (from JDK 5) with class

`java.util.concurrent.atomic.AtomicBoolean`

Operations

```
boolean set();
boolean get();
boolean compareAndSet(boolean expect, boolean update);
boolean getAndSet(boolean newValue);
```

atomically set to value `update` iff current value is `expect`. Return true on success.

sets `newValue` and returns previous value.

How does this work?

- **The JVM bytecode does not offer atomic operations like CAS.**
  [It does, however, support monitors via instructions monitorenter, monitorexit, we will understand this later]
- **But there is a (yet undocumented) class `sun.misc.Unsafe` offering direct mappings from java to underlying machine / OS.**
- **Direct mapping to hardware is not guaranteed –**
  operations on AtomicBoolean are not guaranteed lock-free

Java.util.concurrent.atomic.AtomicInteger

(source: grepcode.com)

```java

package java.util.concurrent.atomic;
import sun.misc.Unsafe;
```

…

Atomically sets the value to the given updated value if the current value == the expected value.

Parameters:

`expect` the expected value
`update` the new value

Returns:

true if successful. False return indicates that the actual value was not equal to the expected value.

```java

    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
```

TASLock in Java

```
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

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

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

Spinlock:

Try to get the lock.

Keep trying until the lock is acquired (return value is false).

unlock
release the lock (set to false)

```java
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

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

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

Simple TAS Spin Lock – Measurement Results

TAS
n = 1, elapsed= 224, normalized= 224
n = 2, elapsed= 719, normalized= 359
n = 3, elapsed= 1914, normalized= 638
n = 4, elapsed= 3373, normalized= 843
n = 5, elapsed= 4330, normalized= 866
n = 6, elapsed= 6075, normalized= 1012
n = 7, elapsed= 8089, normalized= 1155
n = 8, elapsed= 10369, normalized= 1296
n = 16, elapsed= 41051, normalized= 2565
n = 32, elapsed= 156207, normalized= 4881
n = 64, elapsed= 619197, normalized= 9674

- run n threads
- each thread acquires and releases the TASLock a million times
- repeat scenario ten times and add up runtime
- record time per thread

Intel core i7@3.4 GHz, 4 cores + HT

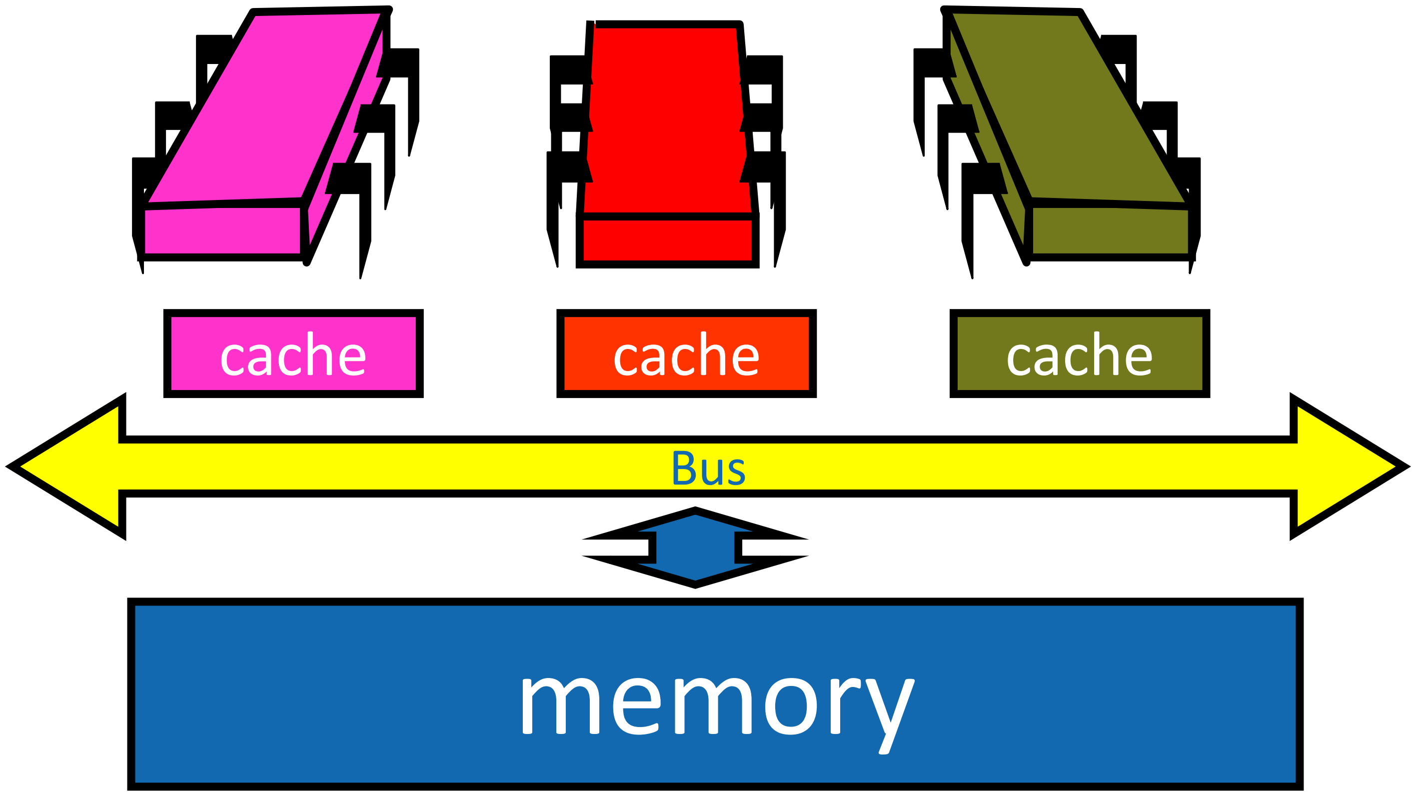
```java
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

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

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

Why?

sequential bottleneck

contention: threads fight for the bus during call of getAndSet()

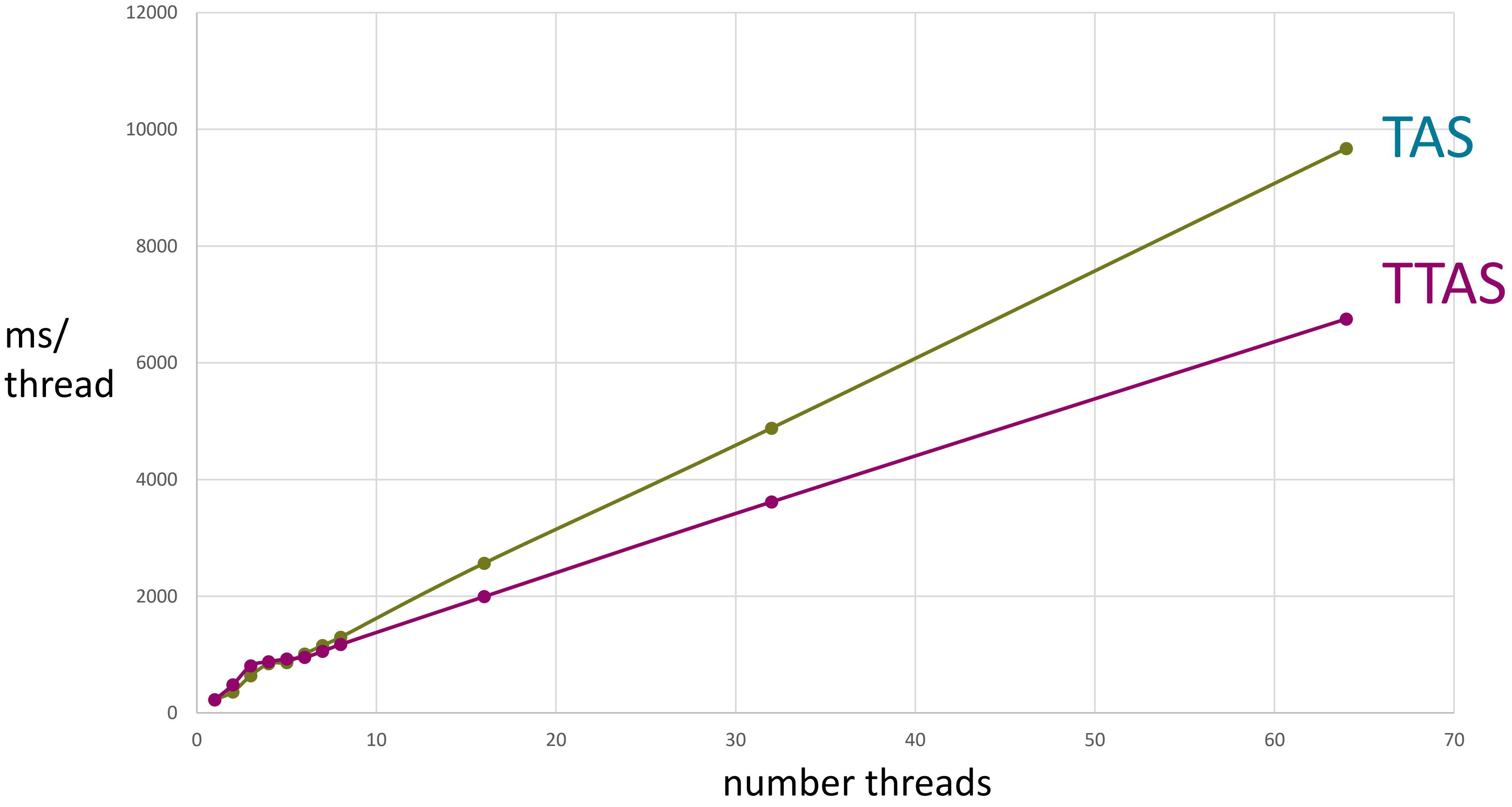
cache coherency protocol invalidates cached copies of the lock variable on other processors



Test-and-Test-and-Set (TATAS) Lock

```
public void lock()
{
    do
        while(state.get()) {}
    while (!state.compareAndSet(false, true));
}

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

Measurement



note that this varies strongly between machines and JVM implementations and even between runs. Take it as a qualitative statement

TATAS does not generalize

- **Example: Double-Checked Locking**

Double-Checked Locking

An Optimization Pattern for Efficiently
Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

Tim Harrison
harrison@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Abstract

This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.

context of concurrency. To illustrate this, consider how the canonical implementation [1] of the Singleton pattern behaves in multi-threaded environments.

The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance [1]. Dynamically allocating Singletons in C++ programs is common since the order of initialization of global static objects in C++ programs is not well-defined and is therefore non-portable. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
  static Singleton *instance (void)
  {
    if (instance_ == 0)
      // Critical section.
      instance_ = new Singleton;

    return instance_;
  }
```

double-checked locking

About 830,000 results (0.27 seconds)

Double-checked locking - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Double-checked_locking
In software engineering, **double-checked locking** (also known as "**double-checked locking** optimization") is a software design pattern used to reduce the ...
Usage in Java - Usage in Microsoft Visual C++ - Usage in Microsoft .NET ...

The "Double-Checked Locking is Broken" Declaration
www.cs.umd.edu/~pugh/java/.../DoubleCheckedLocking.html
Details on the reasons - some very subtle - why **double-checked locking** cannot be relied upon to be safe. Signed by a number of experts, including Sun ...

Double-checked locking and the Singleton pattern
www.ibm.com/developerworks/java/library/j-dcl/index.html
1 May 2002 – **Double-checked locking** is one such idiom in the Java programming language that should never be used. In this article, Peter Haggar ...

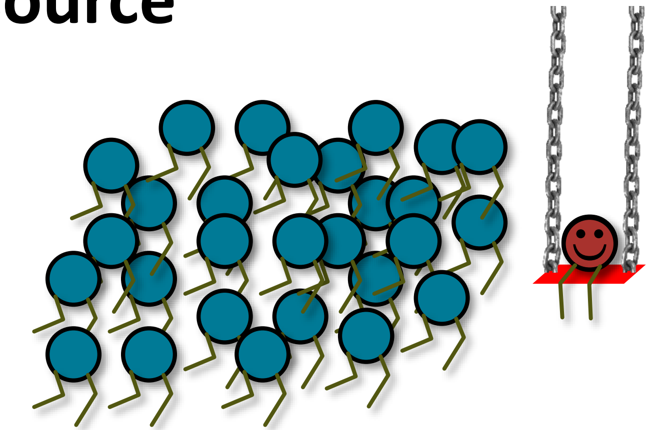
Double-checked locking: Clever, but broken - JavaWorld
www.javaworld.com › Java Development Tools
9 Feb 2001 – Many Java programmers are familiar with the **double-checked locking** idiom, which allows you to perform lazy initialization with reduced ...

[PDF] Double-Checked Locking An Optimization Pattern for Efficiently ...
sunsite.icm.edu.pl/packages/ace/ACE/PDF/DC-Locking.pdf
File Format: PDF/Adobe Acrobat - Quick View
by DC Schmidt - Cited by 14 - Related articles
solve this problem, we present the **Double-Checked Lock**- ing optimization ...
Double-Checked Locking illustrates how changes in under- lying forces (i.e. ...

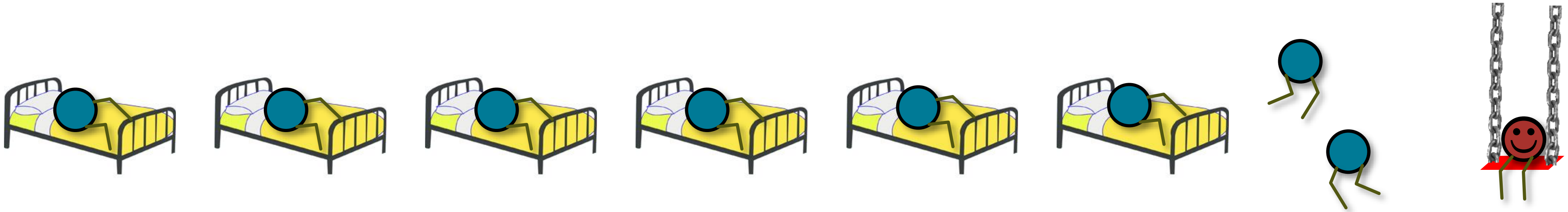
Problem: Memory ordering leads to race-conditions!

TATAS with backoff

Observation
- (too) many threads fight for access to the same resource
- slows down progress globally and locally



Solution
- threads go to sleep with random duration
- increase expected duration each time the resource is not free



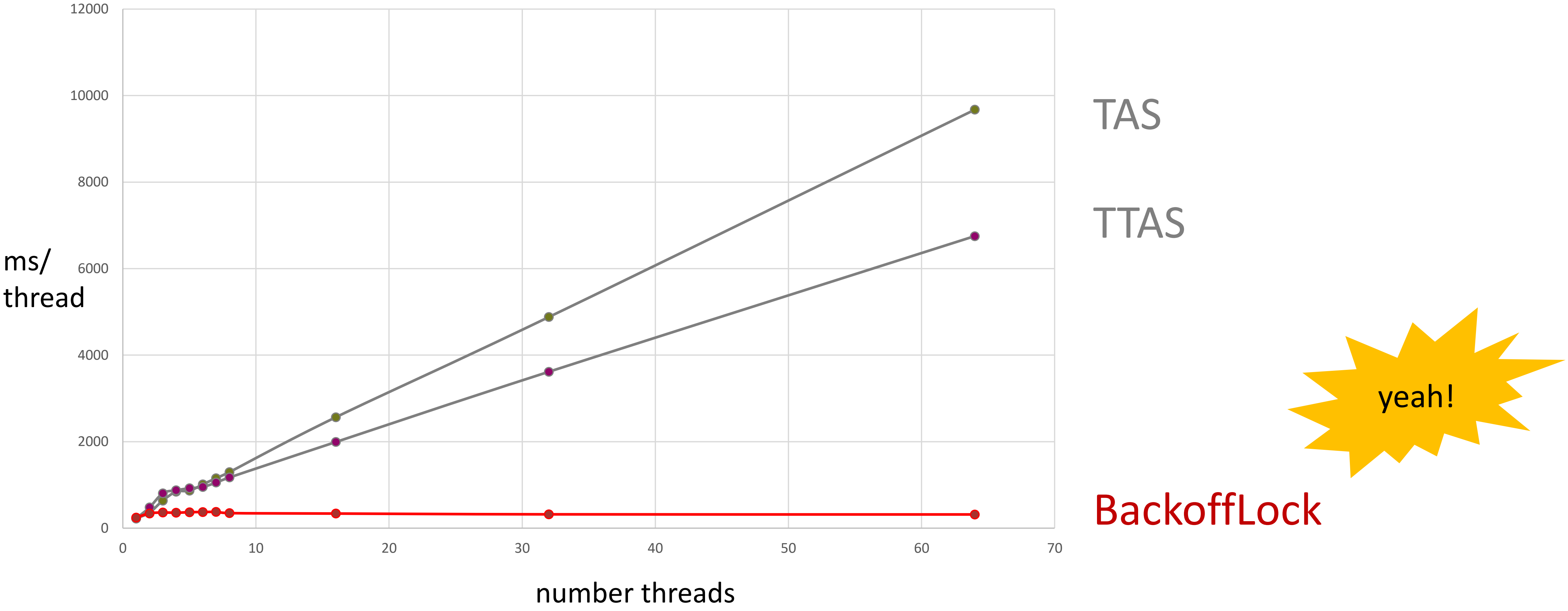
Lock with Backoff

```
public void lock() {
    Backoff backoff = null;
    while (true) {
        while (state.get()) {};        // spin reading only (TTAS)
        if (!state.getAndSet(true))    // try to acquire, returns previous val
            return;
        else { // backoff on failure
            try {
                if (backoff == null)   // allocation only on demand
                    backoff = new Backoff(MIN_DELAY, MAX_DELAY);
                backoff.backoff();
            } catch (InterruptedException ex) {}
        }
    }
}
```

exponential backoff

```
class Backoff
{...
      public void backoff() throws InterruptedException {
            int delay = random.nextInt(limit);
            if (limit < maxDelay) { // double limit if less than max
                  limit = 2 * limit;
            }
            Thread.sleep(delay);
      }
}
```

Measurement



Deadlock

Deadlocks – Motivation

Consider a method to transfer money between bank accounts

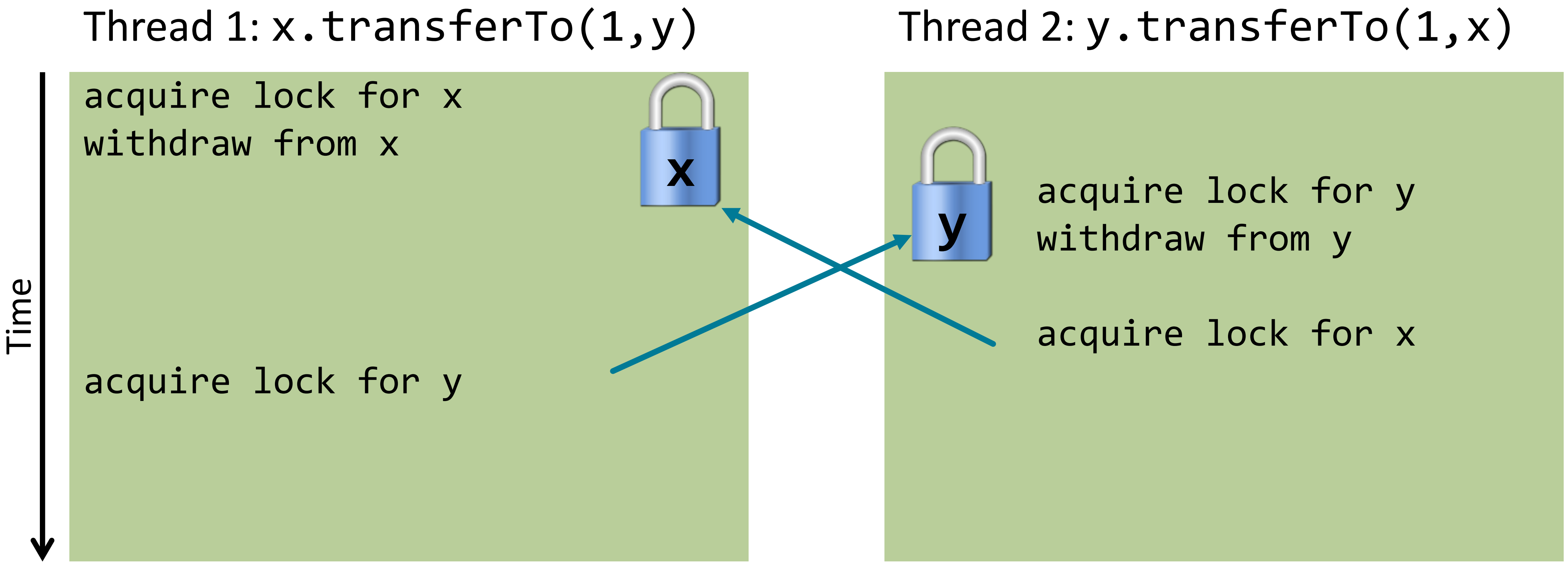
```
class BankAccount {
  …
  synchronized void withdraw(int amount) {…}
  synchronized void deposit(int amount) {…}

  synchronized void transferTo(int amount, BankAccount a) {
    this.withdraw(amount);
    a.deposit(amount);
  }
}
```

Thread aquires second lock in `a.deposit`.
Can this become a problem?

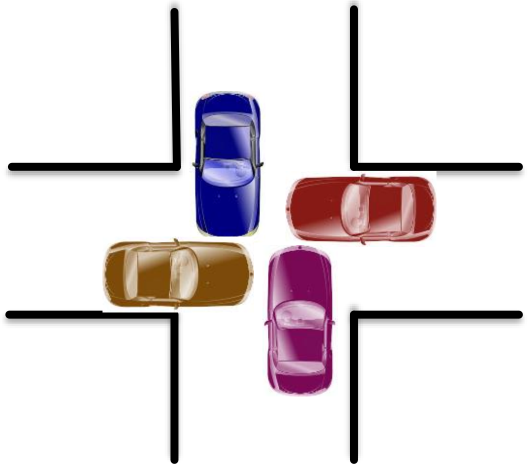
Deadlocks – Motivation

```
class BankAccount {
  …
  synchronized void withdraw(int amount) {…}
  synchronized void deposit(int amount) {…}

  synchronized void transferTo(int amount, BankAccount a) {
    this.withdraw(amount);
    a.deposit(amount);
  }
}
```

Suppose `x` and `y` are instances of class BankAccount



Deadlocks



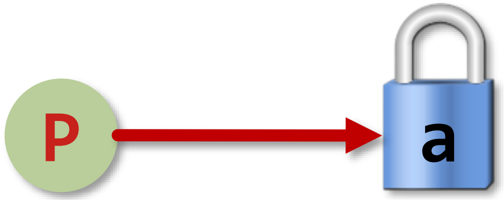
Deadlock: two or more processes are mutually blocked because each process waits for another of these processes to proceed.

Threads and Resources

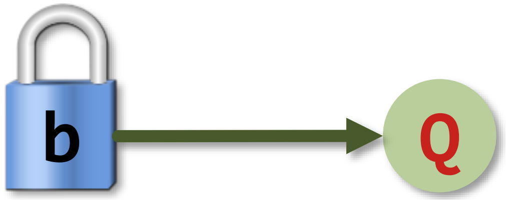
Graphically: Threads 

and Resources (Locks) 

Thread P *attempts to* acquire resource a:

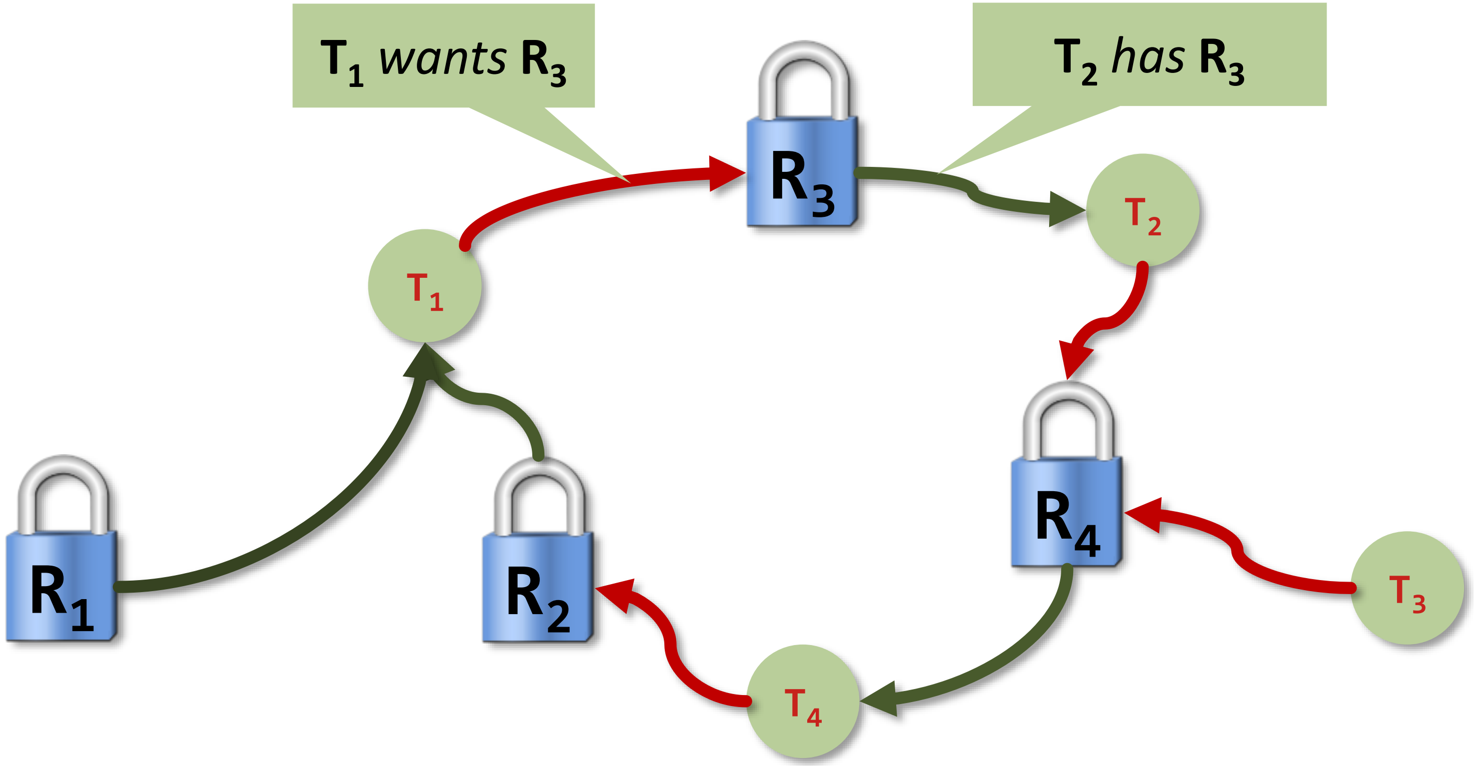


Resource b is *held by* thread Q:



Deadlocks – more formally

A deadlock for threads $T_1 \ldots T_n$ occurs when the directed graph describing the relation of $T_1 \ldots T_n$ and resources $R_1 \ldots R_m$ contains a cycle.



Techniques

***Deadlock detection* in systems is implemented by finding cycles in the dependency graph.**

- **Deadlocks can, in general, not be healed. Releasing locks generally leads to inconsistent state.**

***Deadlock avoidance* amounts to techniques to ensure a cycle can never arise**

- **two-phase locking with retry (release when failed)**
  - Usually in databases where transactions can be aborted without consequence
- **resource ordering**
  - Usually in parallel programming where global state is modified

Back to our example: what can we do?

```
class BankAccount {
  ...
  synchronized void withdraw(int amount) {…}
  synchronized void deposit(int amount) {…}
  ...
  synchronized void transferTo(int amount, BankAccount a) {
    this.withdraw(amount);
    a.deposit(amount);
  }
}
```

Option 1: non-overlapping (smaller) critical sections

```
class BankAccount {
   ...
   synchronized void withdraw(int amount) {…}
   synchronized void deposit(int amount) {…}
   ...
   void transferTo(int amount, BankAccount a) {
     this.withdraw(amount);
     a.deposit(amount);
   }
}
```

Money disappears for a (very short?) moment!
Can we allow such transient inconsistencies?
Very often unacceptable!

Option 2: one lock for all

```
class BankAccount {
   static Object globalLock = new Object();
   // withdraw and deposit protected with globalLock!
   void withdraw(int amount) {…}
   void deposit(int amount) {…}
   ...
   void transferTo(int amount, BankAccount to) {
      synchronized (globalLock) {
         withdraw(amount);
         to.deposit(amount);
      }
   }
}
```

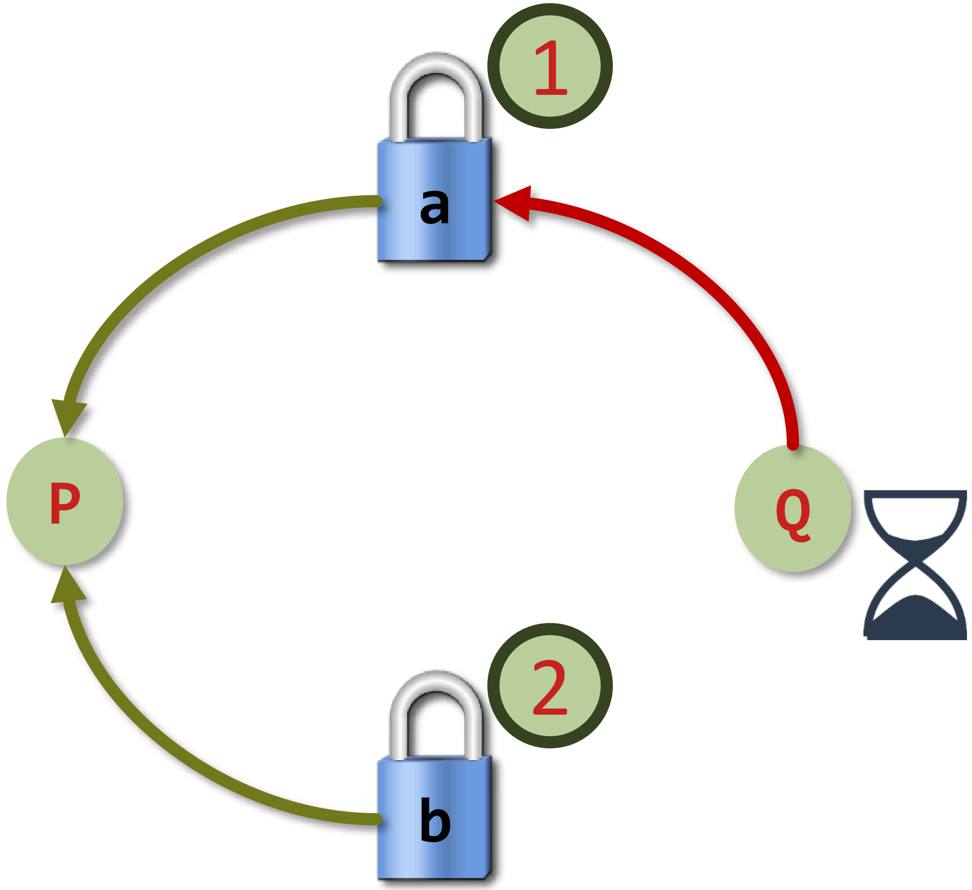
deadlock avoided but no concurrent transfer possible, even not when the pairs of accounts are disjoint.
Often very inefficient!

Option 3: global ordering of resources

```
class BankAccount {
   ...
   void transferTo(int amount, BankAccount to) {
      if (to.accountNr < this.accountNr)
         synchronized(this){
            synchronized(to) {
               withdraw(amount);
               to.deposit(amount);
         }}
      else
         synchronized(to){
            synchronized(this) {
               withdraw(amount);
               to.deposit(amount);
         }}
   }
}
```

Unique global ordering required.
Whole program has to obey this order to avoid cycles.
Code taking only one lock can ignore it.

Ordering of resources



Programming trick

No globally unique order available? Generate it:

```
class BankAccount {
   private static final AtomicLong counter = new AtomicLong();
   private final long index = counter.incrementAndGet();
   ...
   void transferTo(int amount, BankAccount to) {
      if (to.index < this.index)
      ...
   }
}
```

Another (historic) example: from the Java standard library

```
class StringBuffer {
   private int count;
   private char[] value;
   …
   synchronized append(StringBuffer sb) {
      int len = sb.length();
      if(this.count + len > this.value.length)
         this.expand(…);
      sb.getChars(0, len, this.value, this.count);
   }

   synchronized getChars(int x, int y, char[] a, int z) {
      "copy this.value[x..y] into a starting at z"
   }
}
```

Do you find the two problems?

Another (historic) example: from the Java standard library

```
class StringBuffer {
   private int count;
   private char[] value;
   …
   synchronized append(StringBuffer sb) {
      int len = sb.length();
      if(this.count + len > this.value.length)
         this.expand(…);
      sb.getChars(0, len, this.value, this.count);
   }

   synchronized getChars(int x, int y, char[] a, int z) {
      "copy this.value[x..y] into a starting at z"
   }
}
```

Do you find the two problems?

Problem #1:

- Lock for **sb** is not held between calls to `sb.length` and `sb.getChars`
- **sb** could get longer
- Would cause **append** to not append whole string
  - The semantics here can be discussed! Definitely an issue if **sb** got shorter ☺

Problem #2:

- Deadlock potential if two threads try to append "crossing" StringBuffers, just like in the bank-account first example
- `x.append(y); y.append(x);`

Amy Williams, William Thies, and Michael D. Ernst: Static Deadlock Detection for Java Libraries, ECOOP'05 (for deadlock)

Fix?

- **Not easy to fix both problems without extra overheads:**
  - Do not want unique ids on every `StringBuffer`
  - Do not want one lock for all `StringBuffer` objects
- **Actual Java library: initially fixed neither (left code as is; changed javadoc)**
  - Up to clients to avoid such situations with own protocols
- **Today: two classes StringBuffer (claimed to be synchronized) and StringBuilder (not synchronized)**

Perspective

Code like account-transfer and string-buffer append are difficult to deal with for deadlock

1. **Easier case: different types of objects**
   - Can document a fixed order among types
   - Example: “When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock”

2. **Easier case: objects are in an acyclic structure**
   - Can use the data structure to determine a fixed order
   - Example: “If holding a tree node’s lock, do not acquire other tree nodes’ locks unless they are children in the tree”

Significance of Deadlocks

Once understood that (and where) race conditions can occur, with following good programming practice and rules they are relatively easy to cope with.

But the ***Deadlock*** is **the dominant problem** of reasonably complex concurrent programs or systems and is therefore very important to anticipate!

Starvation denotes the repeated but unsuccesful attempt of a recently unblocked process to continue its execution.

Semaphores

Why do we need more than locks?

- **Locks provide means to enforce atomicity via mutual exclusion**

- **They lack the means for threads to communicate about changes**
  - e.g., changes in the state

- **Thus, they provide no order and are hard to use**
  - e.g., if threads A and B lock object X, it is not determined who comes first

- **Example: producer / consumer queues**

Semaphore Edsger W. Dijkstra 1965

Se|ma|phor, das od. der; -s, -e [zu griech. σεμα = Zeichen u. φορος = tragend]:
Signalmast mit beweglichen Flügeln.

Optische Telegrafievorrichtung mit Hilfe von schwenkbaren Signalarmen, Claude Chappe 1792

Semaphore: Semantics

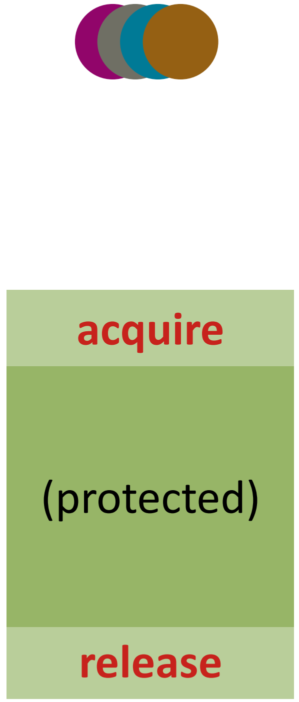
Semaphore: integer-valued abstract data type `S` with some initial value `s`≥0 and the following operations*

```
acquire(S)
{
   wait until S > 0
   dec(S)
}
```

atomic

```
release(S)
{
   inc(S)
}
```

atomic



* Dijkstra called them *P (probeeren)*, *V (vrijgeven)*, also often used: *wait and signal*

Building a lock with a semaphore

mutex = Semaphore(1);

lock mutex := mutex.acquire()

only one thread is allowed into the critical section

unlock mutex := mutex.release()

one other thread will be let in

Semaphore number:

1 → unlocked

0 → locked

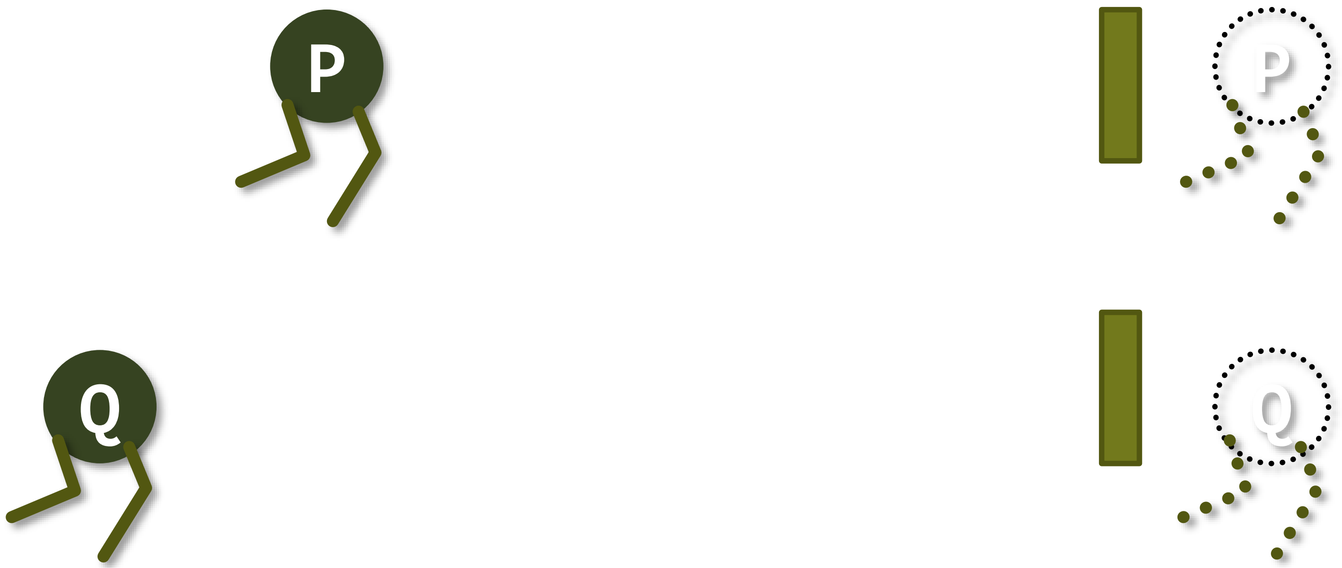
x>0 → x threads will be let into "critical section"

Example: scaled dot product

- **Execute in parallel: $x = (\underline{a}^T * \underline{d}) * z$**
  - a and d are column vectors
  - x, z are scalar
- **Assume each vector has 4 elements**
  - $x = (a_1*d_1 + a_2*d_2 + a_3*d_3 + a_4*d_4) * z$
- **Parallelize on two processors (using two threads A and B)**
  - $x_A = a_1*d_1 + a_2*d_2$
  - $x_B = a_3*d_3 + a_4*d_4$
  - $x = (x_A + x_B) * z$
- **Which synchronization is needed where?**
  - Using locks?
  - Using semaphores?

Rendezvous with Semaphores

- **Two processes P and Q executing code.**
- **Rendezvouz: locations in code, where P and Q wait for the other to arrive. Synchronize P and Q.**



How would you implement this using Semaphores?

Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)

Semaphores `P_Arrived` and `Q_Arrived`

	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	...	...
rendezvous	?	?
post	...	..

Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)

Semaphores `P_Arrived` and `Q_Arrived`

	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `?`	`acquire(P_Arrived)` `?`
post	`...`	`...`

Rendezvous with Semaphores

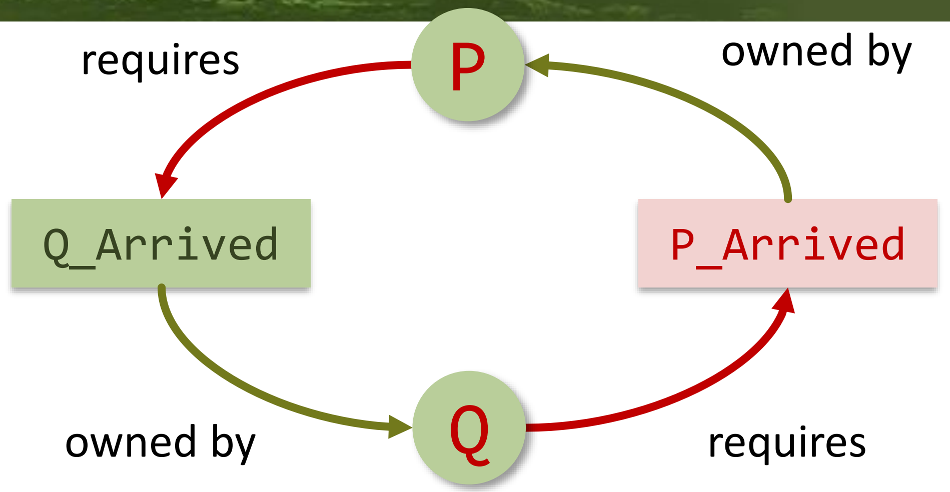
Synchronize Processes P and Q at one location (Rendezvous)

Semaphores `P_Arrived` and `Q_Arrived`

Dou you find the problem?

	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`acquire(Q_Arrived)` `release(P_Arrived)`	`acquire(P_Arrived)` `release(Q_Arrived)`
post	`...`	`...`

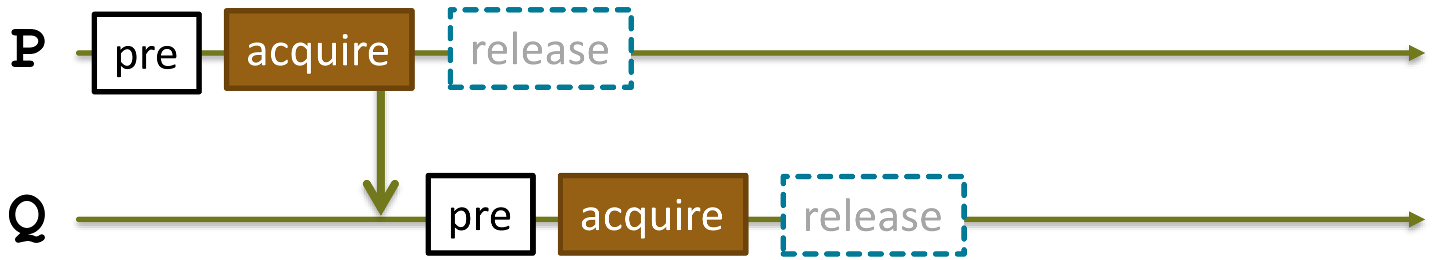
Deadlock



	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`acquire(Q_Arrived)` `release(P_Arrived)`	`acquire(P_Arrived)` `release(Q_Arrived)`
post	`...`	`...`

Rendezvous with Semaphores

Wrong solution with Deadlock



Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)

Assume Semaphores `P_Arrived` and `Q_Arrived`

	P	**Q**
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `acquire(Q_Arrived)`	`acquire(P_Arrived)` `release(Q_Arrived)`
post	`...`	`..`

Implementing Semaphores without Spinning (blocking queues)

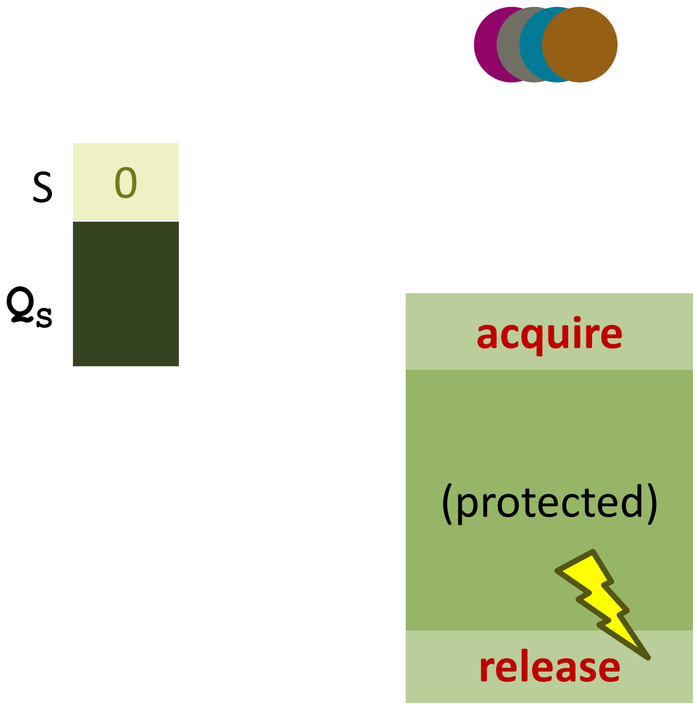
Consider a process list Q_S associated with semaphore S

acquire(S)

atomic
```
{if S > 0 then
     dec(S)
 else
     put(Q_S, self)
     block(self)
 end }
```

release(S)

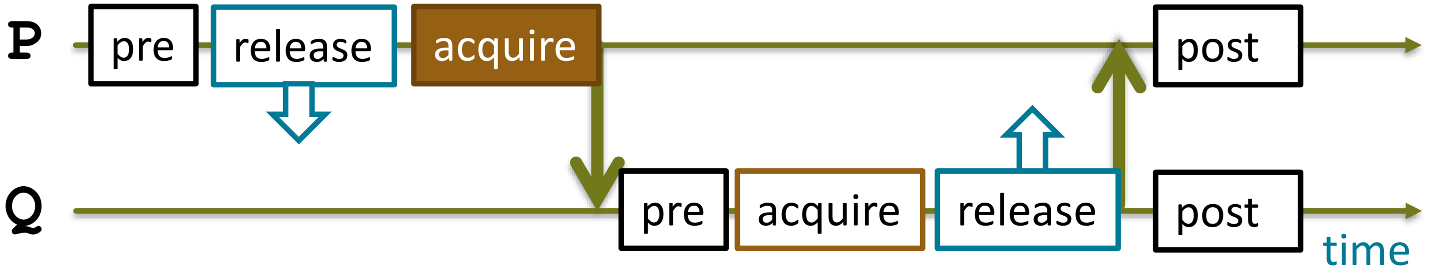
atomic
```
{if Q_S == ∅ then
     inc(S)
 else
     get(Q_S, p)
     unblock(p)
 end }
```



Scheduling Scenarios

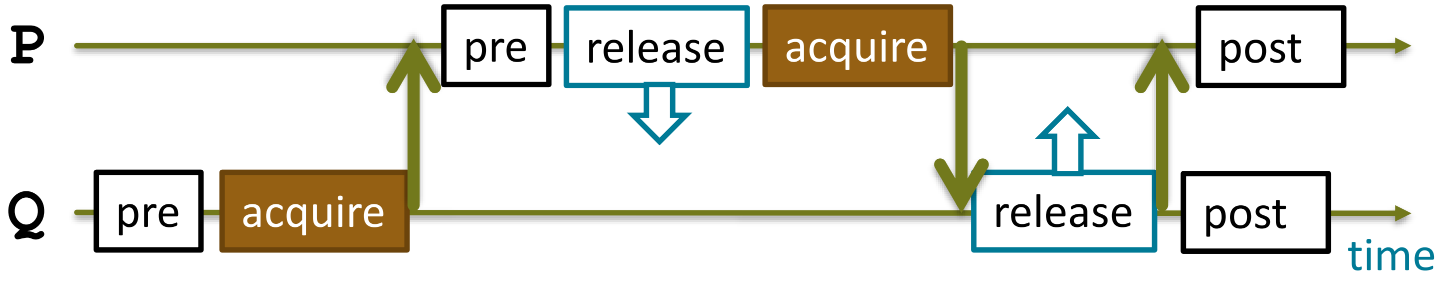
	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `acquire(Q_Arrived)`	`acquire(P_Arrived)` `release(Q_Arrived)`
post	`...`	`..`

P first



release signals (arrow)
acquire may wait (filled box)

Q first



Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)

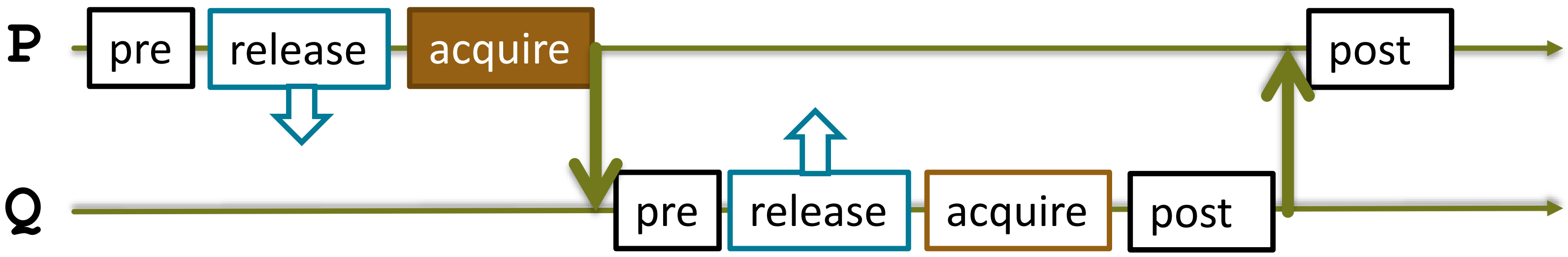
Assume Semaphores `P_Arrived` and `Q_Arrived`

	P	**Q**
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `acquire(Q_Arrived)`	`release(Q_Arrived)` `acquire(P_Arrived)`
post	`...`	`..`

That’s even better.

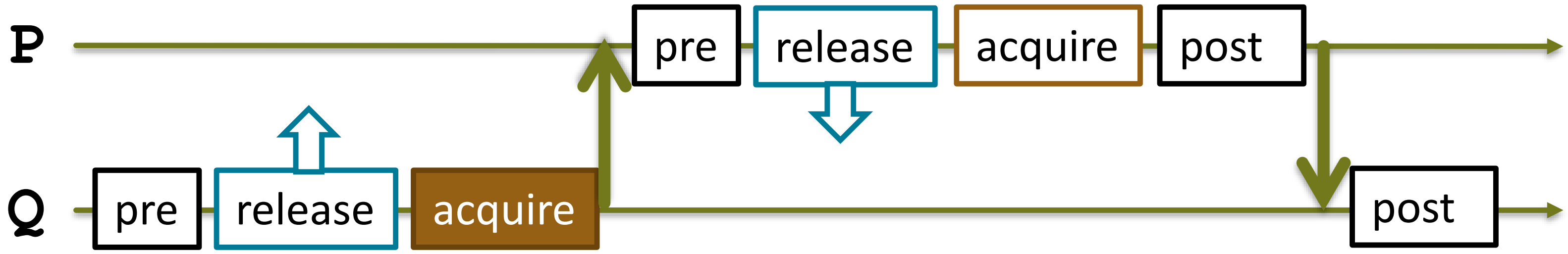
	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `acquire(Q_Arrived)`	`release(Q_Arrived)` `acquire(P_Arrived)`
post	`...`	`..`

P first



Q first

release signals (arrow)
acquire may wait (filled box)



Back to our dot-product

- **Assume now vectors with 1 million entries on 10,000 threads**
  - Very common! (over the weekend, we ran 57 Pflop/s on 27,360 GPUs)
  - How would you implement that?
  - Semaphores, locks?

- **Time for a higher-level abstraction!**
  - Supporting threads in bulk-mode
    Move in lock-step
  - And enabling a “bulk-synchronous parallel” (BSP) model
    The full BSP is more complex (supports distributed memory)

A Bridging Model for Parallel Computation

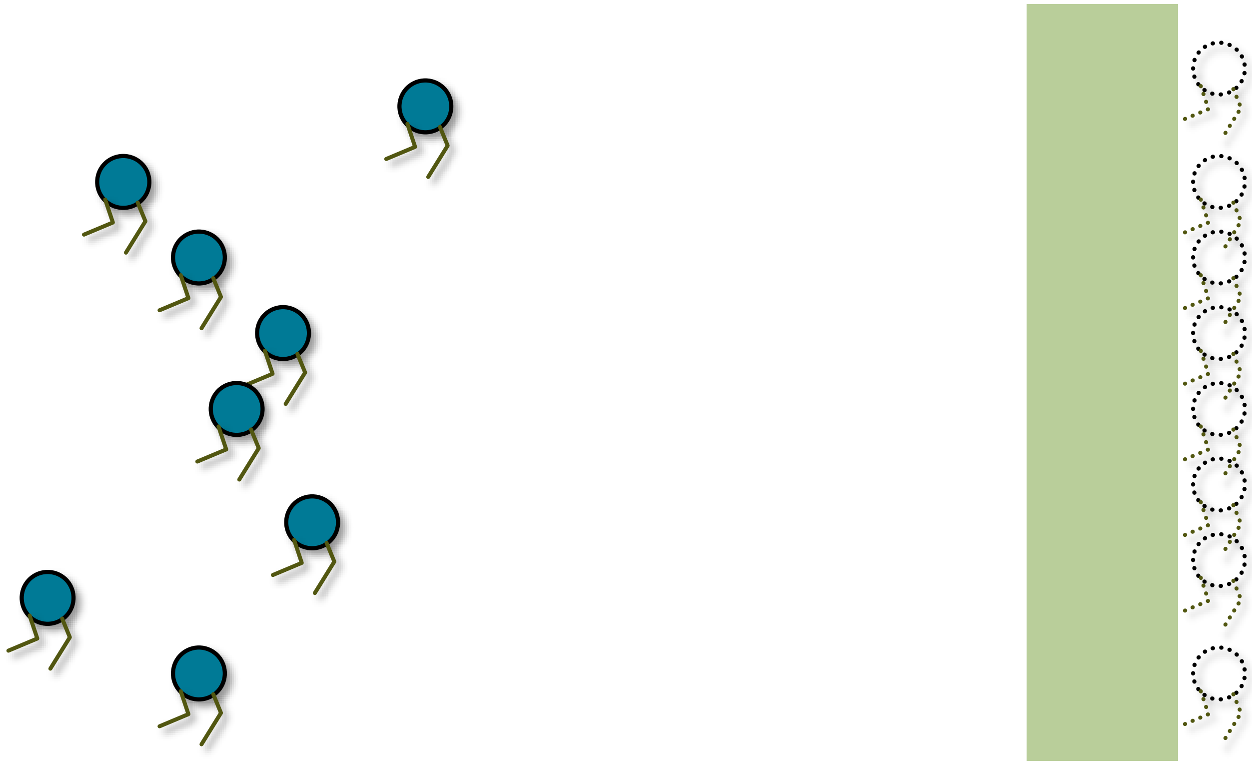
The success of the von Neumann model of sequential computation is attributable to the fact that it is an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware. The author argues that an analogous bridge between software and hardware is required for parallel computation if that is to become as widely used. This article introduces the bulk-synchronous parallel (BSP) model as a candidate for this role, and gives results quantifying its efficiency both in implementing high-level language features and algorithms, as well as in being implemented in hardware.

Leslie G. Valiant

Barriers

Barrier

Synchronize a number of processes.



How would you implement this using Semaphores?

Barrier – 1st try

Synchronize a number (n) of processes.

Semaphore `barrier`. Integer `count`.

	P1	P2	...	Pn
init	`barrier = 0; volatile count = 0`			
pre	`...`			

barrier	`count++`
	`if (count==n) release(barrier)`
	`acquire(barrier)`
		←	←	←

post	`...`			

WRONG

Race Condition !

Some wait forever!

Barrier

Synchronize a number (n) of processes.

Semaphore `barrier`. Integer `count`.

	P1
init	`barrier = 0; volatile c`
pre	`...`

barrier	`count++`
	`if (count==n) release(barrier)`
	`acquire(barrier)`

post	`...`

WRONG

Invariants

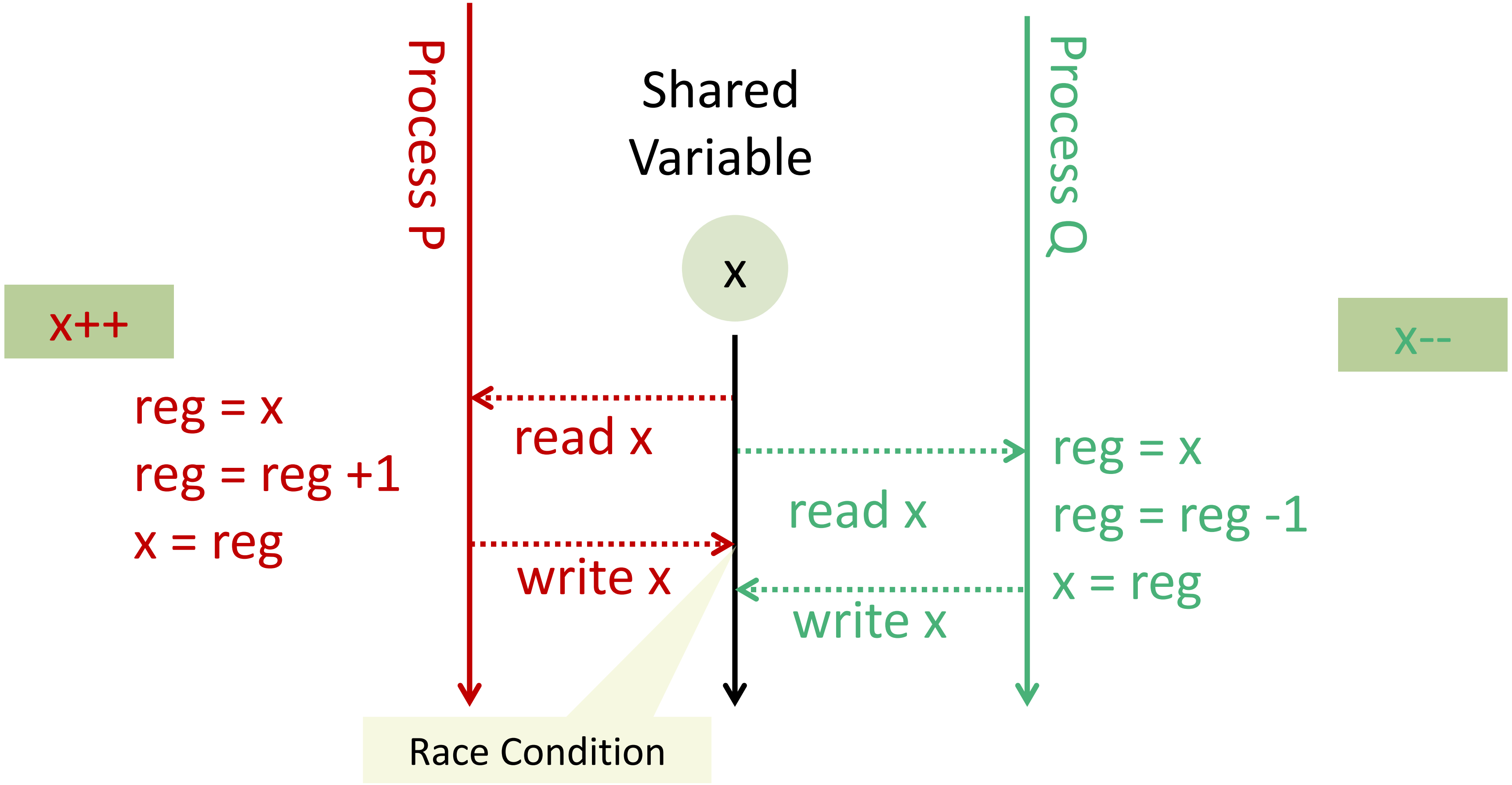
«Each of the processes eventually reaches the acquire statement"

«The barrier will be opened if and only if all processes have reached the barrier"

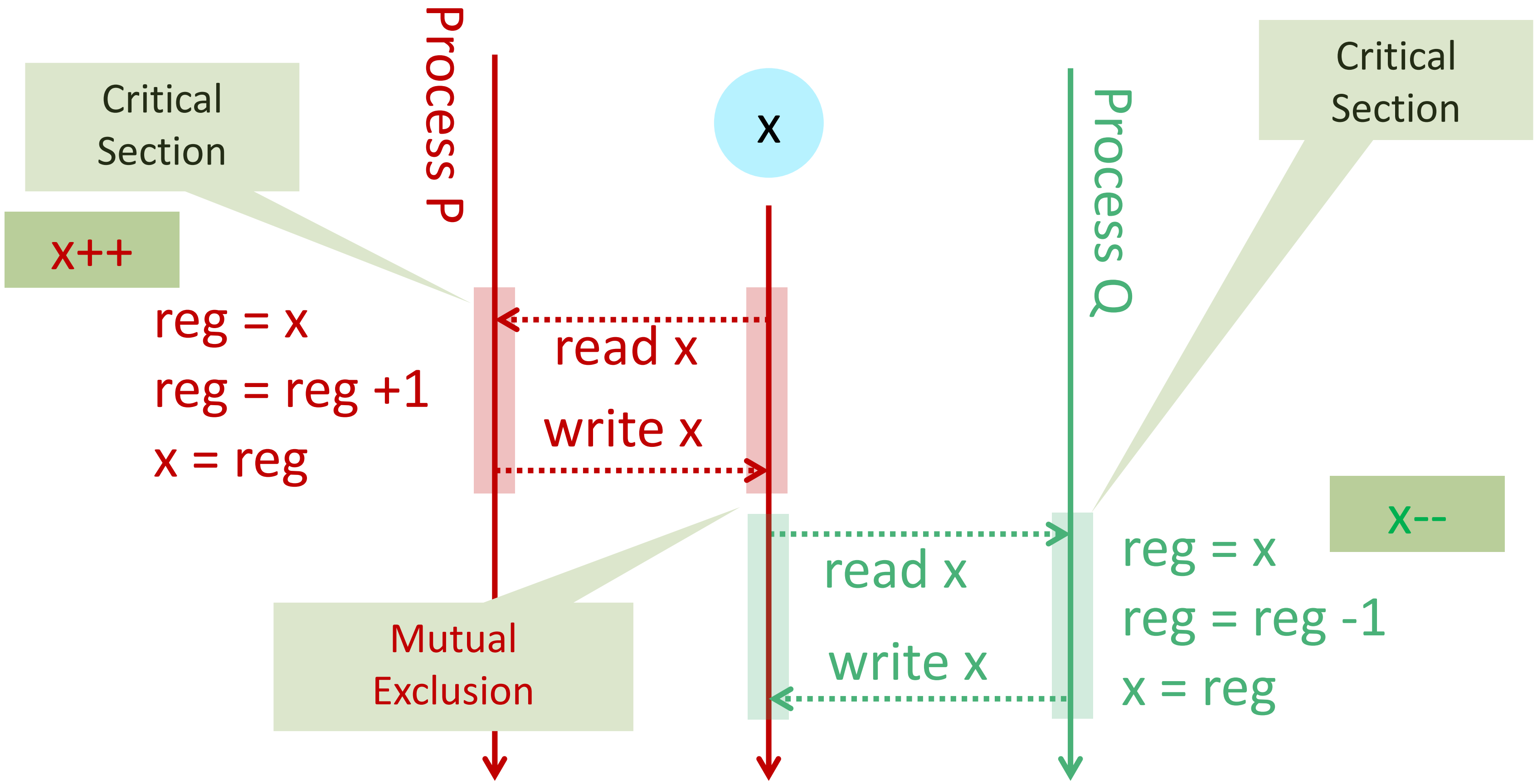
«count provides the number of processes that have passed the barrier" (violated)

«when all processes have reached the barrier then all waiting processes can continue" (violated)

Recap: Race Condition



With Mutual Exclusion



Barrier

Synchronize a number (n) of processes.

Semaphores `barrier`, `mutex`. Integer `count`.

	P1	P2	...	Pn
init	`mutex = 1; barrier = 0; count = 0`			
pre	`...`			

barrier	`acquire(mutex)`
	`  count++`
	`release(mutex)`
	`if (count==n) release(barrier)`
	`acquire(barrier)`
	`release(barrier)`
	turnstile
		
		←
			
			←
				
				←

post	`...`			

Reusable Barrier. 1st trial.

	P1	...	Pn
init	mutex = 1; barrier = 0; count = 0		
pre	...		

barrier	acquire(mutex)
	  count++
	release(mutex)
	if (count==n) release(barrier)
	
	acquire(barrier)
	release(barrier)
	
	acquire(mutex)
	  count--
	release(mutex)
	if (count==0) acquire(barrier)
		←	←

post	...		

Dou you see the problem?

Race Condition !

Race Condition !

Reusable Barrier. 1st trial.

	P1	...	Pn
init	`mutex = 1; barrier = 0; count = 0`		
pre	`...`		

barrier	`acquire(mutex)`
	`  count++`
	`release(mutex)`
	`if (count==n) release(barrier)`
	
	`acquire(barrier)`
	`release(barrier)`
	
	`acquire(mutex)`
	`  count--`
	`release(mutex)`
	`if (count==0) acquire(barrier)`
			←

post	`...`		

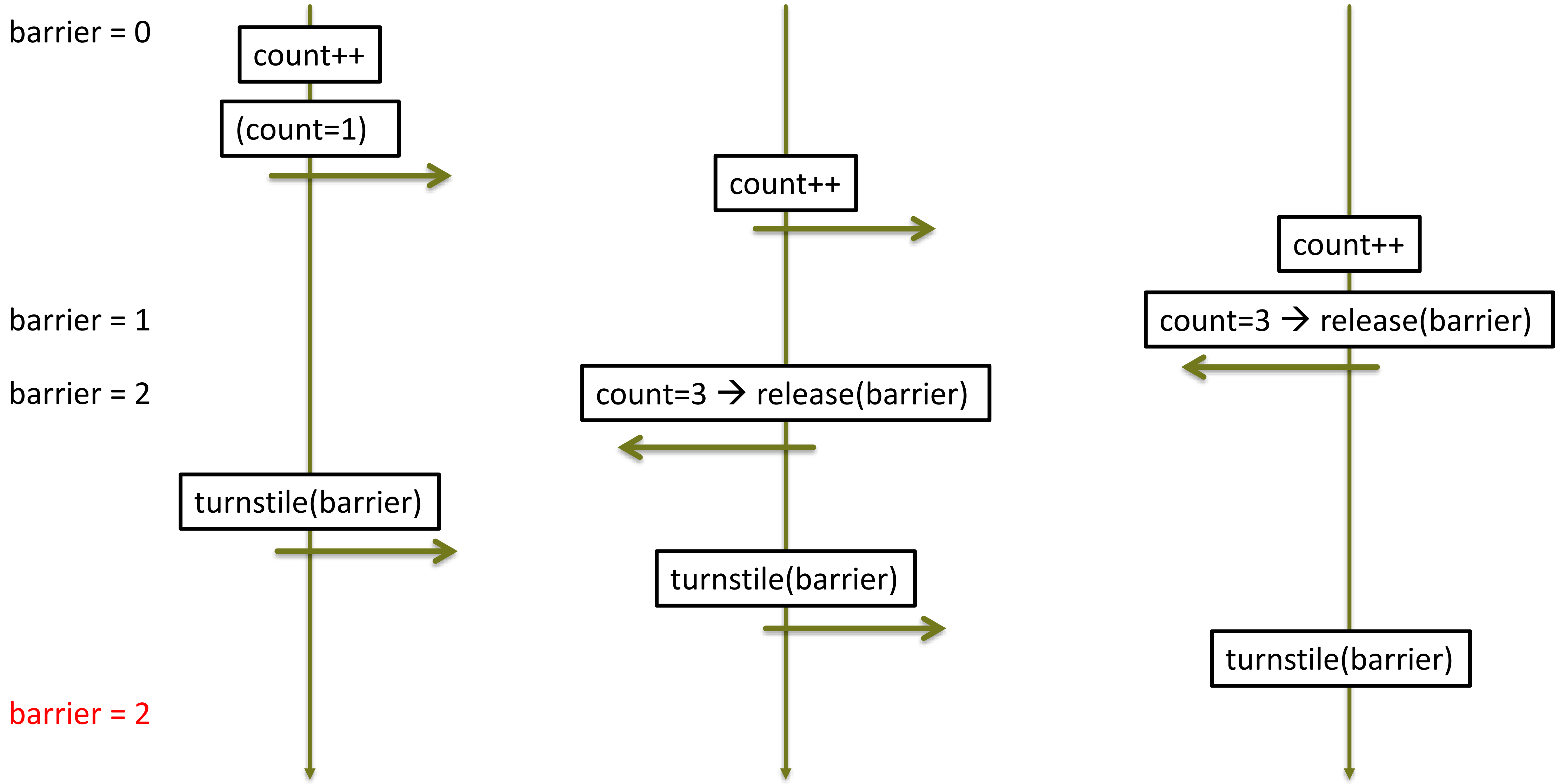
Invariants

«Only when all processes have reached the turnstyle it will be opened the first time"

«When all processes have run through the barrier then count = 0"

«When all processes have run through the barrier then barrier = 0" (violated)

Illustration of the problem: scheduling scenario



Reusable Barrier. 2nd trial.

	P1	...	Pn
init	`mutex = 1; barrier = 0; count = 0`		
pre	`...`		

barrier	`acquire(mutex)`
	`count++`
	`if (count==n) release(barrier)`
	`release(mutex)`
	
	`acquire(barrier)`
	`release(barrier)`
	
	`acquire(mutex)`
	`count--`
	`if (count==0) acquire(barrier)`
	`release(mutex)`
		←	←

post	`...`		

Dou you see the problem?

Process can pass other processes!

Reusable Barrier. 2nd trial.

	P1	...	Pn
init	`mutex = 1; barrier = 0; count = 0`		
pre	`...`		

barrier	`acquire(mutex)`
	`  count++`
	`  if (count==n) release(barrier)`
	`release(mutex)`
	
	`acquire(barrier)`
	`release(barrier)`
	
	`acquire(mutex)`
	`  count--`
	`  if (count==0) acquire(barrier)`
	`release(mutex)`		

post	`...`		

Invariants

«When all processes have passed the barrier, it holds that barrier = 0"

« Even when a single process has passed the barrier, it holds that barrier = 0» (violated)

Solution: Two-Phase Barrier

init

```
mutex=1; barrier1=0; barrier2=1; count=0
```

barrier

```
acquire(mutex)
  count++;
  if (count==n)
     acquire(barrier2); release(barrier1)
release(mutex)

acquire(barrier1); release(barrier1);
// barrier1 = 1 for all processes, barrier2 = 0 for all processes
acquire(mutex)
  count--;
  if (count==0)
      acquire(barrier1); release(barrier2)
signal(mutex)

acquire(barrier2); release(barrier2)
// barrier2 = 1 for all processes, barrier1 = 0 for all processes
```

Of course, this is very slow in practice, see http://www.spiral.net/software/barrier.html for a specialized fast barrier for x86!

Lesson Learned ?

- **Semaphore, Rendevouz and Barrier:**
- **Concurrent programming is prone to errors in reasoning.**
- **A naive approach with trial and error is close-to impossible.**
- **Ways out:**
  - Identify invariants in the problem domain, ensure they hold for your implementation
  - Identify and apply established patterns
  - Use known good libraries (like in the Java API)

Summary

Locks are not enough: we need methods to wait for events / notifications

Semaphores

Rendezvous and Barriers

Next lecture:

Producer-Consumer Problem

Monitors and condition variables