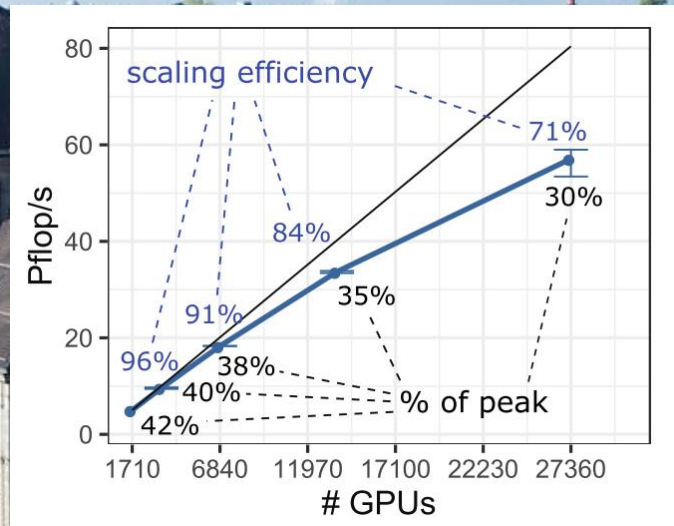
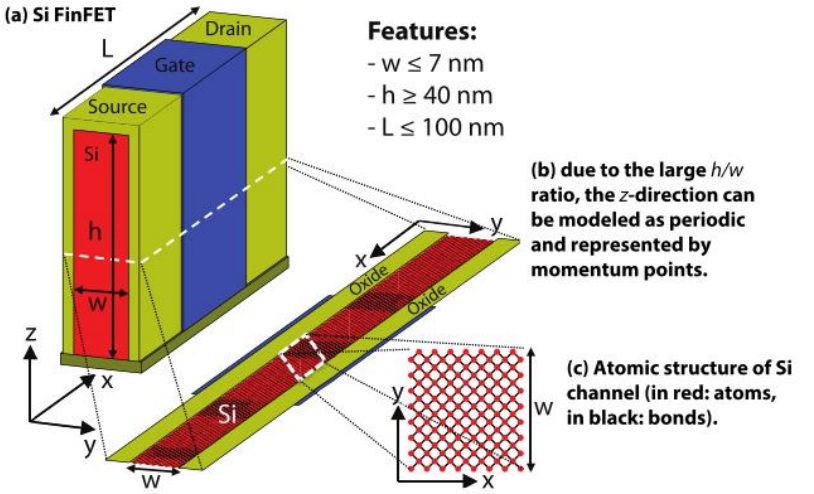
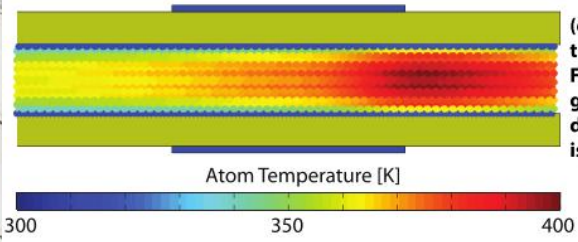
TORSTEN HOEFLER

Parallel Programming, Spring 2021, Lecture 16:
Spinlocks, Deadlocks, Semaphores







(d) Atomically resolved temperature within the FinFET in (a-b), when a gate-to-source and drain-to-source voltage is applied.

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

Some months later

Gordon Bell Award 2019

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Hewlett Packard Enterprise

NVIDIA

ANNOUNCING
THE WORLD'S FASTEST SUPERCOMPUTER FOR AI

20 Exaflops of AI

Powered by NVIDIA Grace CPU and Next Generation NVIDIA GPU

HPC and AI for Scientific and Commercial Apps

Advance Weather, Climate, and Material Science

NVIDIA GTC'21 (https://www.youtube.com/watch?v=eAn_oiZwUXA at 01:03:30)

Learning goals for today

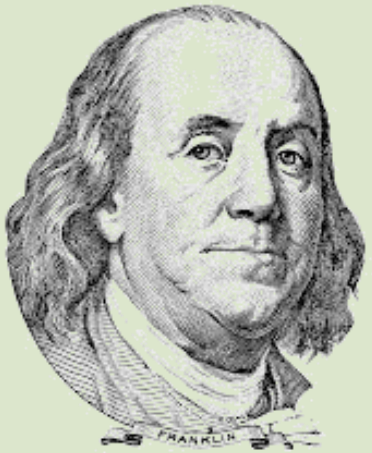
- **So far:**
  - Bad interleavings and data races and why they happen
  - Memory ordering and how we formalize it to drive proofs
  - Implementation of locks using atomic or safe registers (Peterson lock)

- **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

Recap last lecture by a short quiz

- **Please participate in the zoom poll!**



“Tell me and I forget, teach me and I may remember, involve me and I learn.”

More concise than Decker: Peterson Lock (recap)

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

We want to prove ...

that the Peterson Lock satisfies mutual exclusion

and that it is starvation free

§ 1.[7] [1] Das kantonale Steueramt vollzieht das Erbschafts- und Schenkungssteuergesetz (ESchG) vom 28. September 1986[4], soweit nachfolgend nichts Abweichendes geregelt ist.

[2] Die Finanzdirektion entscheidet über Rekurse gemäss §§ 61 Abs. 2 und 64 Abs. 2 ESchG[4].

§ 2.[6] Es gelten sinngemäss:

a. § 119 des Steuergesetzes[2] über den Ausstand,

b. §§ 2–15, § 21 Abs. 1 und 2 sowie §§ 23–25 der Verordnung zum Steuergesetz[3].

How?

Requires some notation first.

Events and precedence

Threads produce a sequence of events (cf. actions)

P produces events $p_0, p_1, \ldots$

e.g., p_1 = "flag[P] = true"

programs usually consist of loops, therefore we might need to count occurences

j-th occurence of event i in thread P: p_i^j

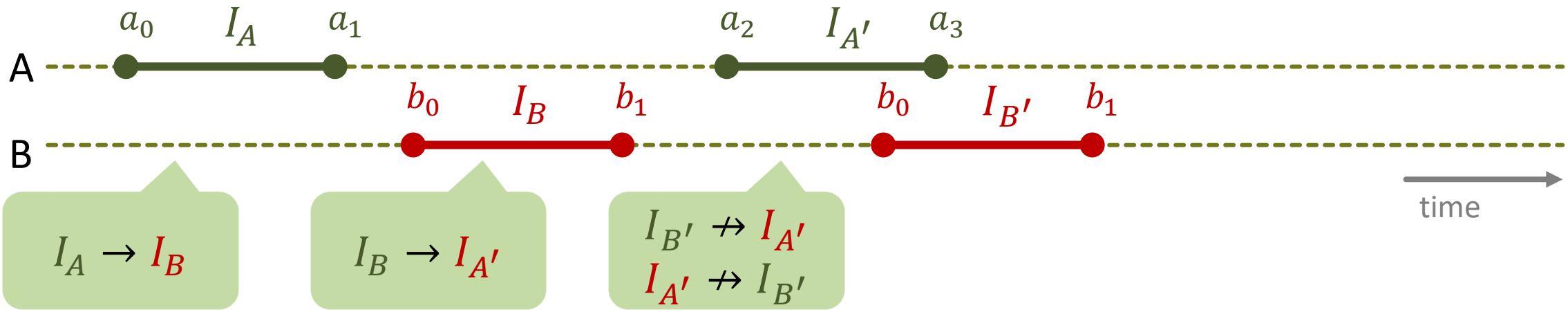
e.g., p_5^3 **= "flag[P] = false" in the third iteration**

Precedence relation: we write $a \rightarrow b$ when a occurs before b.

Note that the precedence relation "→" is a total order for events.

Intervals

(a_0, a_1): **interval of events a_0, a_1 with $a_0 \rightarrow a_1$**
With $I_A = (a_0, a_1)$ and $I_B = (b_0, b_1)$ we write $I_A \rightarrow I_B$ if $a_1 \rightarrow b_0$



we say "I_A ***precedes*** I_B" **and** "$I_{B'}$ **and** $I_{A'}$ **are *concurrent*"**

Atomic register

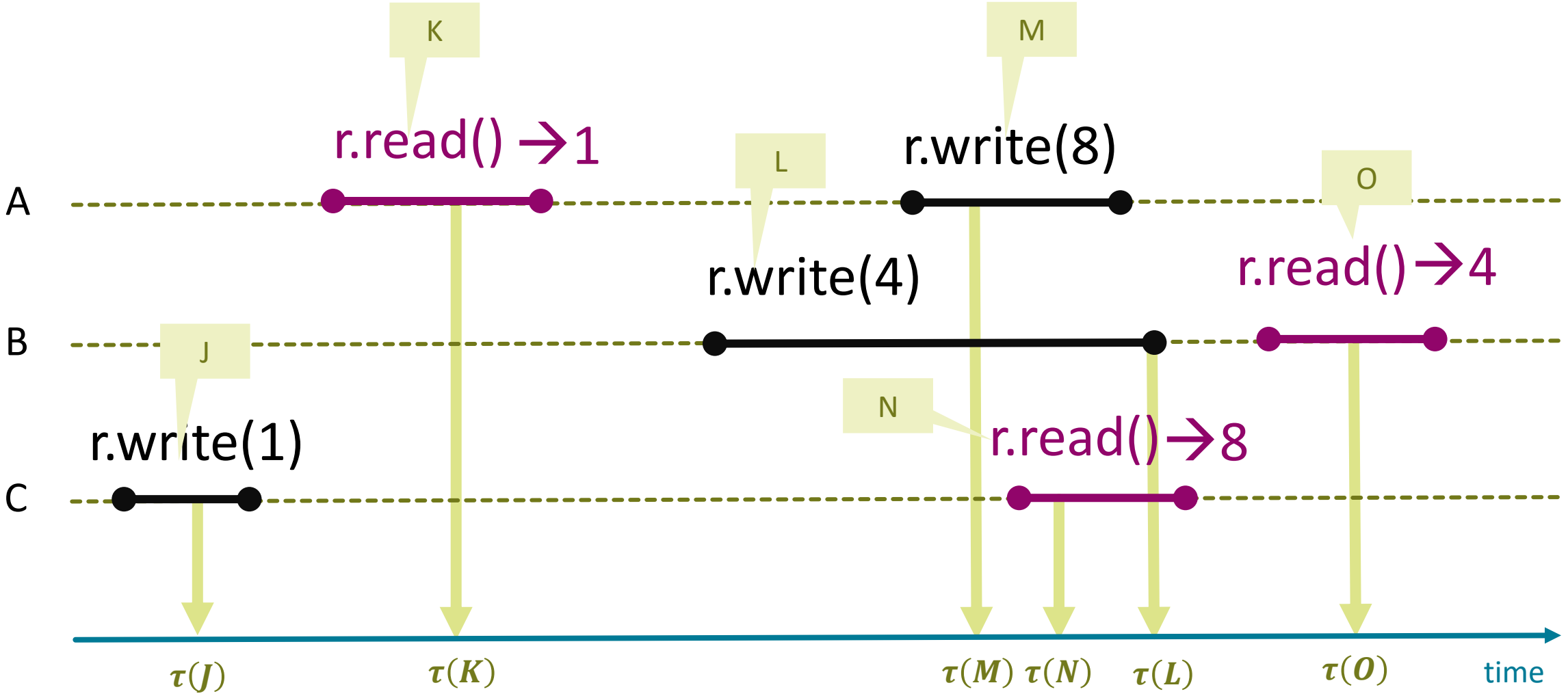
Register: basic memory object, can be shared or not
i.e., in this context register ≠ register of a CPU

Register *r* : operations *r.read()* and *r.write(v)*

Atomic Register:

- **An invocation *J* of *r.read* or *r.write* takes effect at a single point $\tau(J)$ in time**
- **$\tau(J)$ always lies between start and end of the operation *J***
- **Two operations *J* and *K* on the same register always have a different effect time $\tau(J) \neq \tau(K)$**
- **An invocation *J* of *r.read()* returns the value *v* written by the invocation *K* of *r.write(v)* with closest preceding effect time $\tau(K)$**

Example


K
M
r.read() →1
r.write(8)
L
O
A
r.write(4)
r.read()→4
B
J
N
r.write(1)
r.read()→8
C
τ(J)
τ(K)
τ(M)
τ(N)
τ(L)
τ(O)
time

Atomic register

Assumptions for atomic registers justify to treat operations on them as events taking place at a single point in time.

We will use this in the following proofs.

Note that even with atomic registers there can still be non-determinism of programs because nothing is said about the order of effect times for concurrent operations.

Proof: Mutual exclusion (Peterson)

```
flag[P] = true
victim = P
while (flag[Q] && victim == P){}
CS_P
flag[P] = false
```

By contradiction: assume concurrent CS_P and CS_Q [A]

Assume without loss of generality:

$$W_Q(victim=Q) \rightarrow W_P(victim=P)\ [B]$$

From the code:

A + C ⇒ must read false

B ⇒ must read P **[C]**

$$W_P(flag[P]=true) \rightarrow W_P(victim = P) \rightarrow R_P(flag[Q]) \rightarrow R_P(victim) \rightarrow CS_P$$

"write of P"

transitivity of "→ "
⇒ must read true

$$W_Q(flag[Q]=true) \rightarrow W_Q(victim = Q) \rightarrow R_Q(flag[P]) \rightarrow R_Q(victim) \rightarrow CS_Q$$

"read of Q"

Proof: Freedom from starvation

```
flag[P] = true
victim = P
while (flag[Q] && victim == P){}
CS_P
flag[P] = false
```

By (exhaustive) contradition

Assume without loss of generality that P runs forever in its lock (while) loop, waiting until `flag[Q]==false` or `victim != P`.

Possibilities for Q:

stuck in nonCS

⇒ flag[Q] = false and P can continue. Contradiction.

repeatedly entering and leaving its CS

⇒ sets victim to Q when entering.

Now victim cannot be changed ⇒ P can continue. Contradiction.

stuck in its lock loop waiting until `flag[P]==false` or `victim != Q`.

But `victim == P` and `victim == Q` cannot hold at the same time. Contradiction.

Peterson in Java

```
class PetersonLock
{
        volatile boolean flag[] = new boolean[2];
        volatile int victim;

        public void Acquire(int id)
        {
                flag[id] = true;
                victim = id;
                while (flag[1-id] && victim == id);
        }

        public void Release(int id)
        {
                flag[id] = false;
        }
}
```

Volatile reference to an array and not an array of volatile variables!
This example may work in practice.
However, for a correct program we need to use to use Java's **AtomicInteger** and **AtomicIntegerArray**.

The Filter Lock

Extension of Peterson's lock to n processes

Every thread *t* knows its level in the filter *level[t]*

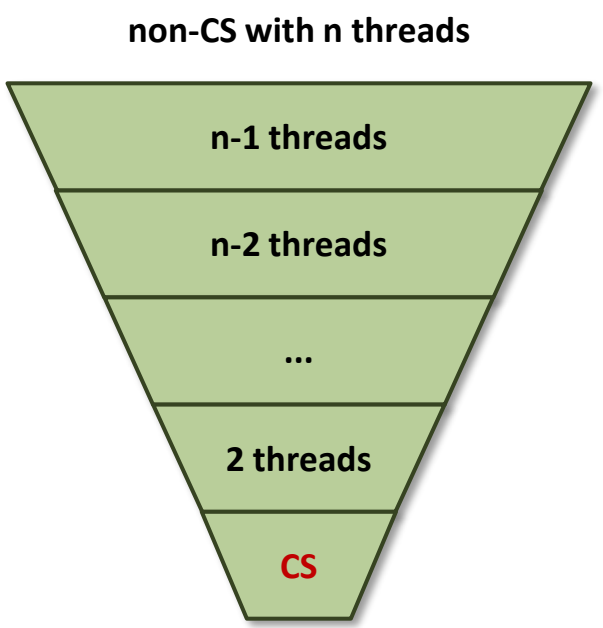
In order to enter CS, a thread has to elevate all levels.

For each level, we use Peterson's mechanism to filter at most one thread, if other threads are at higher level.

For every level *l* there is one victim *victim[l]* that has to let others pass in case of conflicts.

The Filter Lock

```
int[] level(#threads), int[] victim(#threads)



lock(me) {
  for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
  }
}

unlock(me) {
  level[me] = 0;
}
```



Other threads are at same or higher level

And I have to wait

FilterLock in Java

```
import java.util.concurrent.atomic.AtomicIntegerArray;
class FilterLock{
   AtomicIntegerArray level;
   AtomicIntegerArray victim;
   volatile int n;

   FilterLock(int n) {
      this.n = n;
      level = new AtomicIntegerArray(n);
      victim = new AtomicIntegerArray(n);
   }
   ...
```

FilterLock in Java

```
...
   // ∃k ≠ me: level[k] >= i (lev)
   boolean Others(int me, int lev) {
      for (int k = 0; k < n; ++k)
         if (k != me && level.get(k) >= lev) return true;
      return false;
   }
   public void Acquire(int me) {
      for (int lev = 1; lev < n; ++lev) {
         level.set(me, lev);
         victim.set(lev, me);
         while(me == victim.get(lev) && Others(me,lev));
      }
   }
   public void Release(int me) {
      level.set(me, 0);
   }
}
```

Again: I (as a thread) can make progress if
(a) Another thread wants to enter my level or
(b) No more threads are in front of me
This works because there are at most n threads in the system.

Fairness

Divide lock implementation (preprotocol) into two parts

- **doorway interval D: finite number of steps**
- **waiting interval W: unbounded number of steps**

A lock algorithm is first-come-first-served when for two processes A and B it holds that

If $D_A^j \rightarrow D_B^k$ **then** $CS_A^j \rightarrow CS_B^k$

The Filter Lock

satisfies mutual exclusion

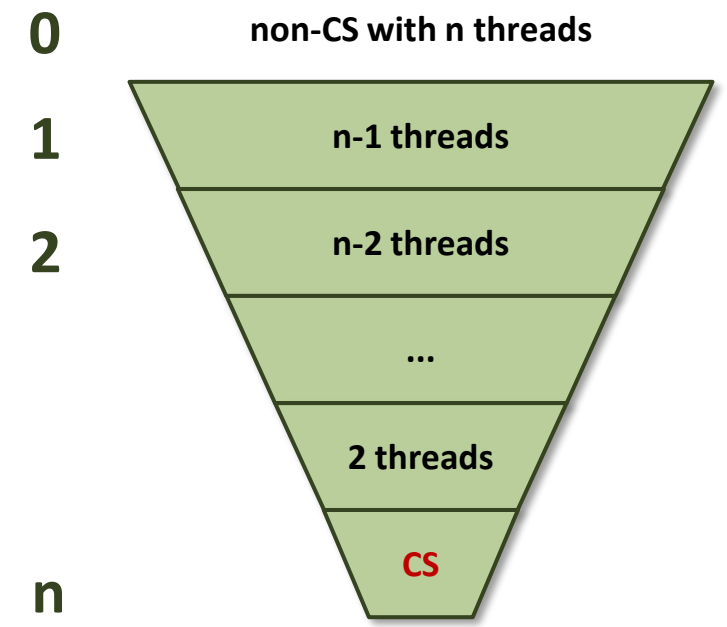
is deadlock free (how to prove?)

is starvation free (how to prove?)

but: is it also fair?

no: the filter lock is not first-come-first-serve

What else is bad about this lock?



A small detour: Safe and Regular Registers

Question:

- **Is it possible to construct mutual exclusion with non-atomic registers?**

Surprisingly: yes

- **It is possible with registers fulfilling the weakest possible conditions that appear to be still useful in a concurrent setup.**

Weakening atomic registers – «the safe SWMR register»

Register r: basic memory object, can be shared or not,
operations *r.read()* and *r.write(v)*.

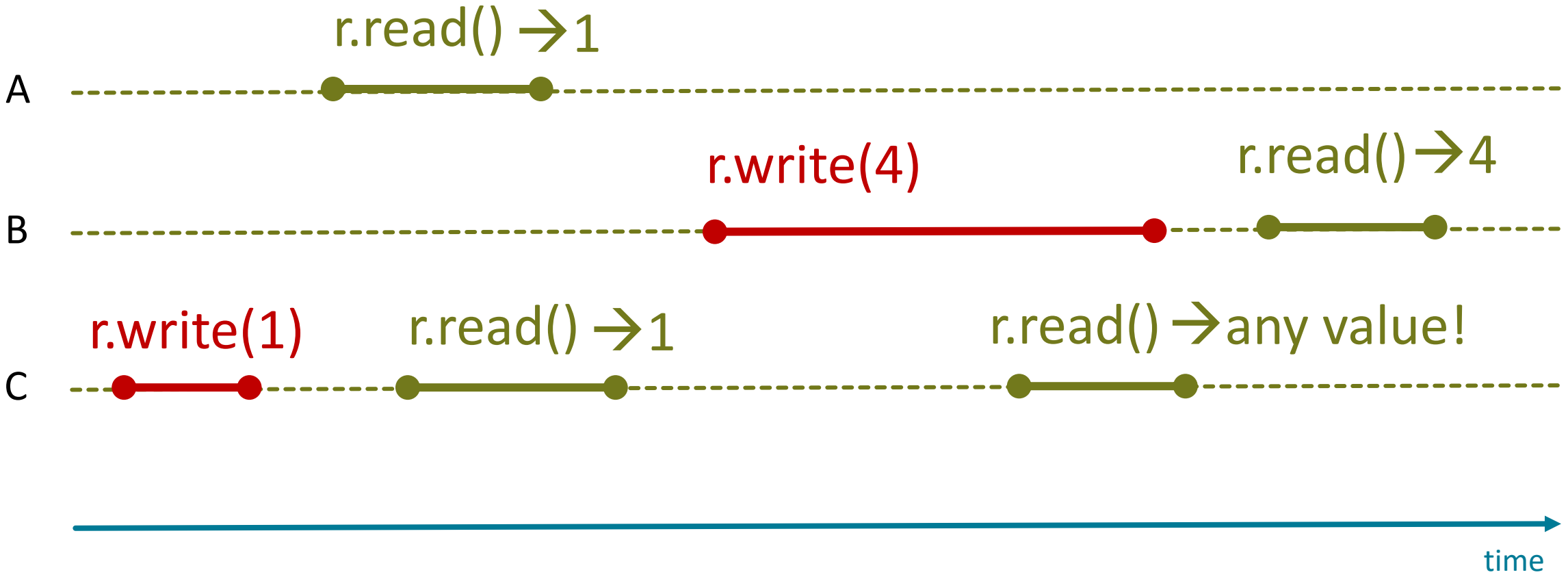
SWMR (Single Writer Multiple Reader): only one concurrent write but multiple concurrent reads allowed.

***Safe* Register**

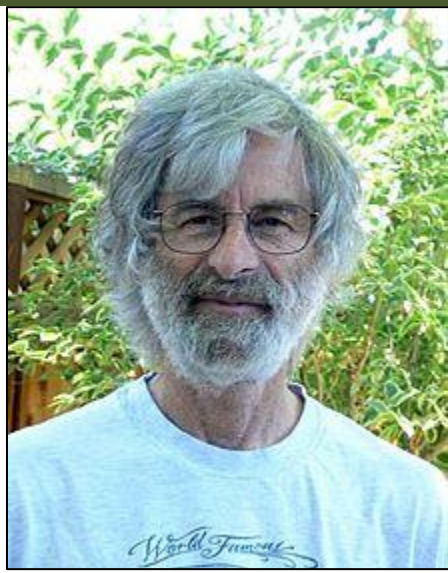
- **any read not concurrent with a write returns the current value of r**
- **any read concurrent with a write can return *any value* of the domain of r**
  if any read concurrent with writes can only return a value of one of the values (previous, new) then the register is called *regular*

The notion "safe" is historically motivated but actually misleading.

Example


r.read() →1
A
r.write(4)
r.read() →4
B
r.write(1)
r.read() →1
r.read() →any value!
C
time

Mutual Exclusion for n processes: Bakery Algorithm (1974)



Lamport, Turing award 2013

A process is required to take a numbered ticket

with value greater than all outstanding tickets

CS Entry: Wait until ticket number is lowest

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 with different access requirements

- **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 the … 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 memory location. If equal, copy the second operand to the first 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

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

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.

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

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

- if the address has the Shared memory attribute, marks the physical address as exclusive access for the executing process
- causes the execut

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»

Syntax

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

where:

<cond> Is the co ed in *The condition* sed.

<Rd> Specifie

<Rn> Specifies the register containing the address.

Architecture version

Version 6 and above.

STREX

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.

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»

Syntax

STREX{<cond>} <R

where:

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

<Rd> S eturned is:

0 if the operation updates memory

1 if the operation fails to update memory.

From the ARM Architecture Reference Manual

Hardware support for atomic operations

Typical instructions

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

Test-And-Set (TAS)

Example TSL register, flag (Motorola 68000)

Compare-And-Swap (CAS)

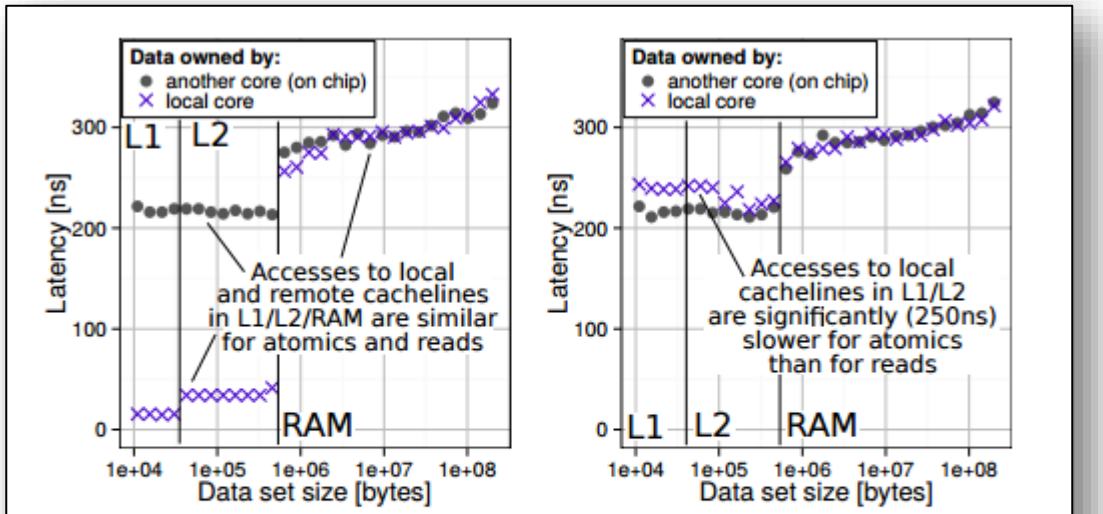
Example: LOCK CMPXCHG (Intel x86)

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? (for experts)

- 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

Example: java.util.concurrent.atomic.AtomicInteger (for experts)

```
35
36    package java.util.concurrent.atomic;
37    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.

```
133
134       public final boolean compareAndSet(int expect, int update) {
135           return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
136       }
```

(source: grepcode.com)

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)

Simple TAS Spin Lock – Measurement Results

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

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

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

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

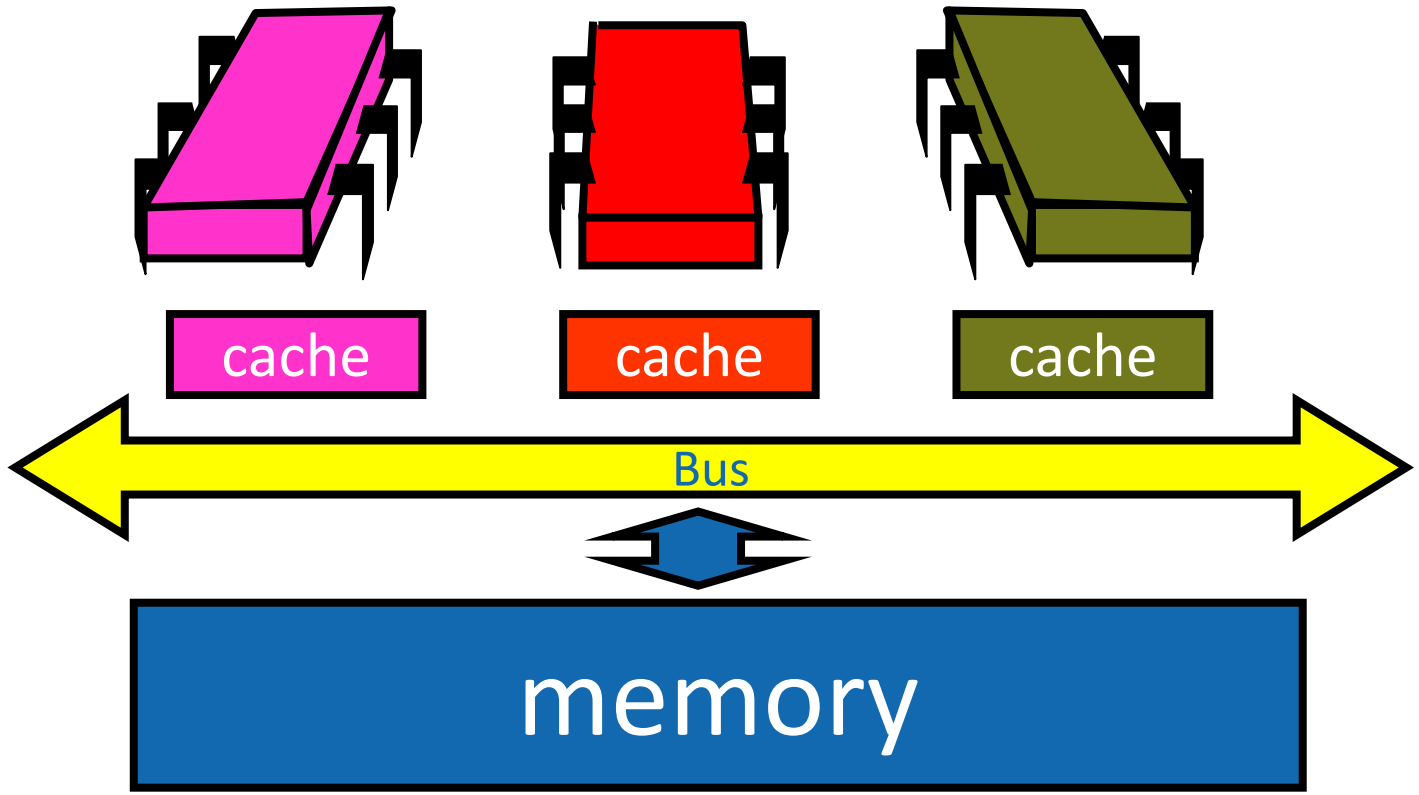
```
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

    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()

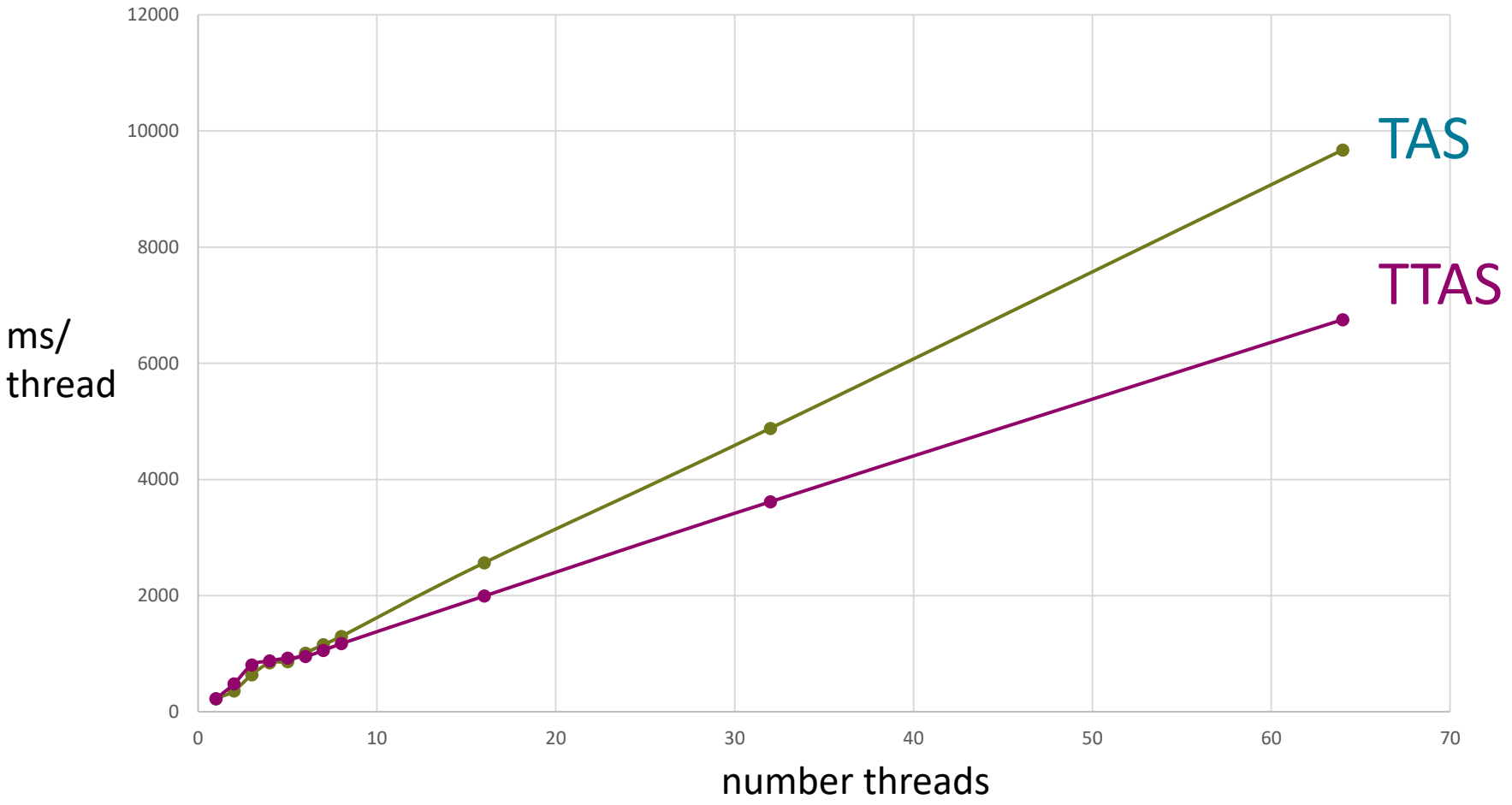
cache coherency protocol invalidates cached copies of the lock variable on other processors



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);
}
```

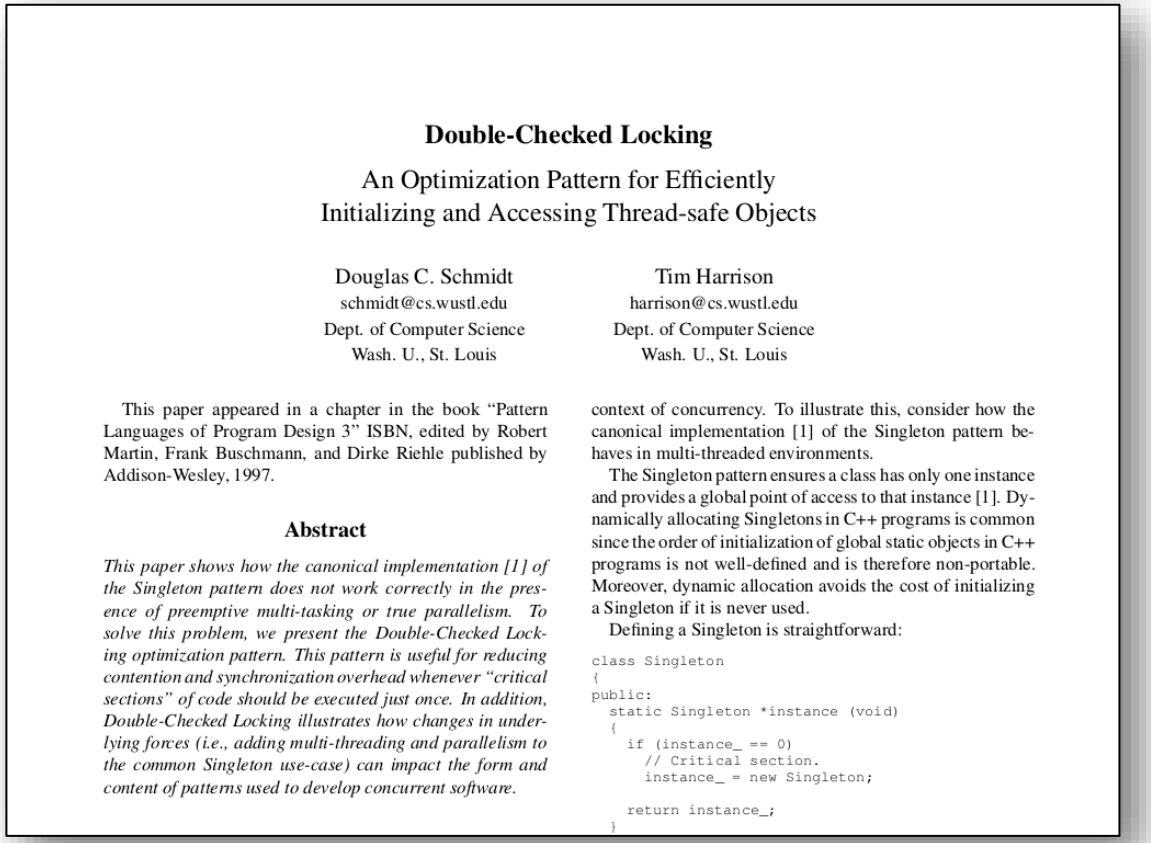
Measurement



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

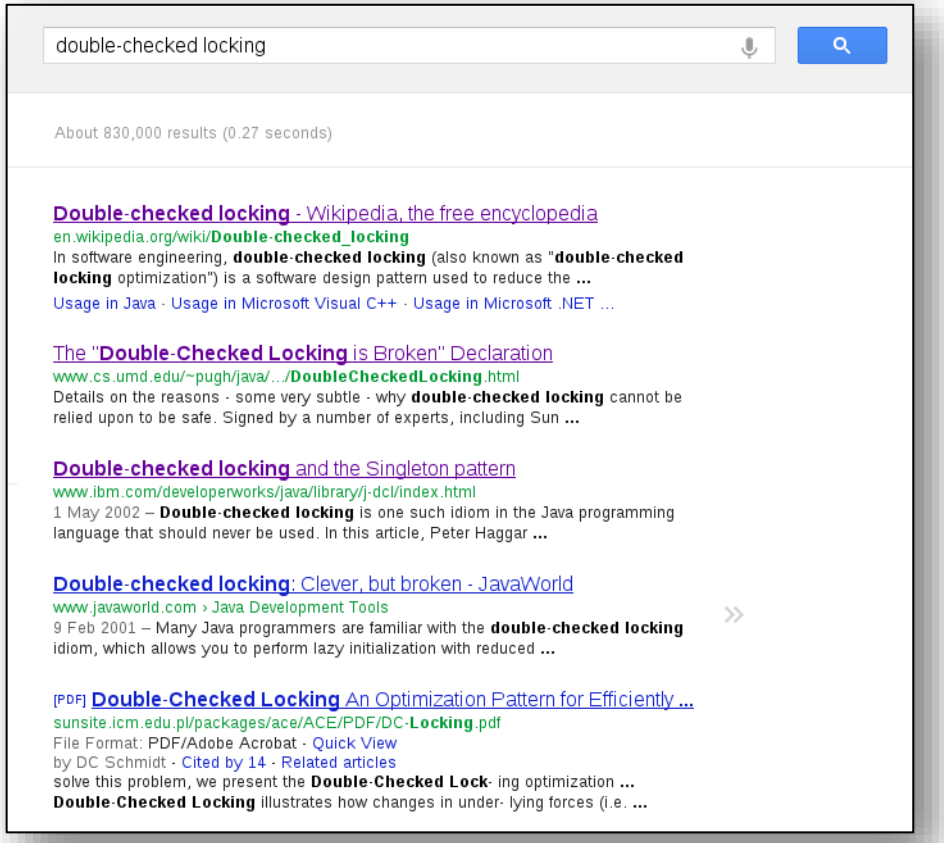
This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and 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_;
  }
```

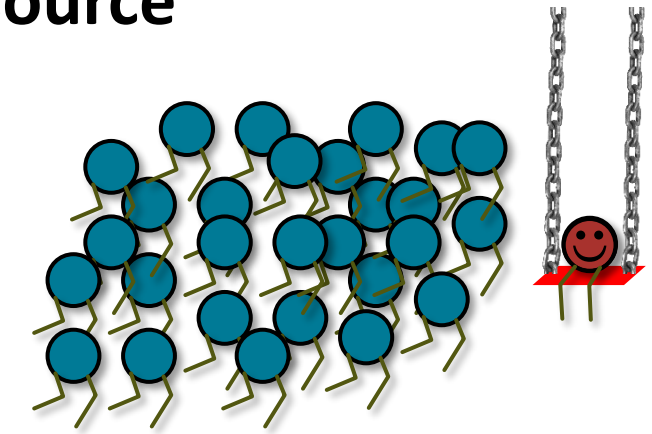


Problem: Memory ordering leads to race-conditions!

TATAS with backoff

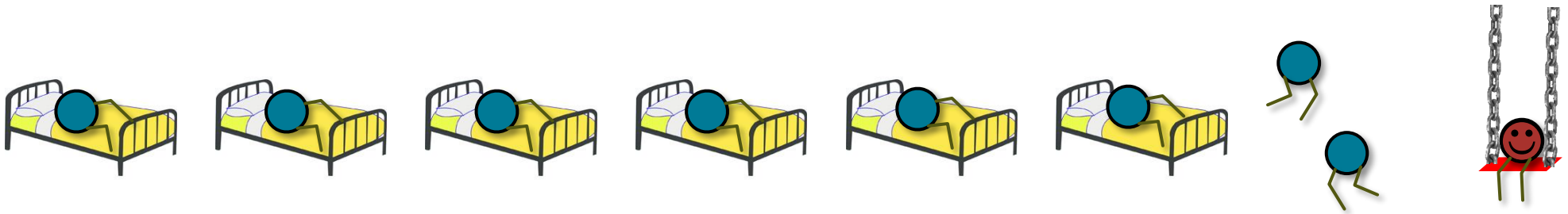
Observation

- (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



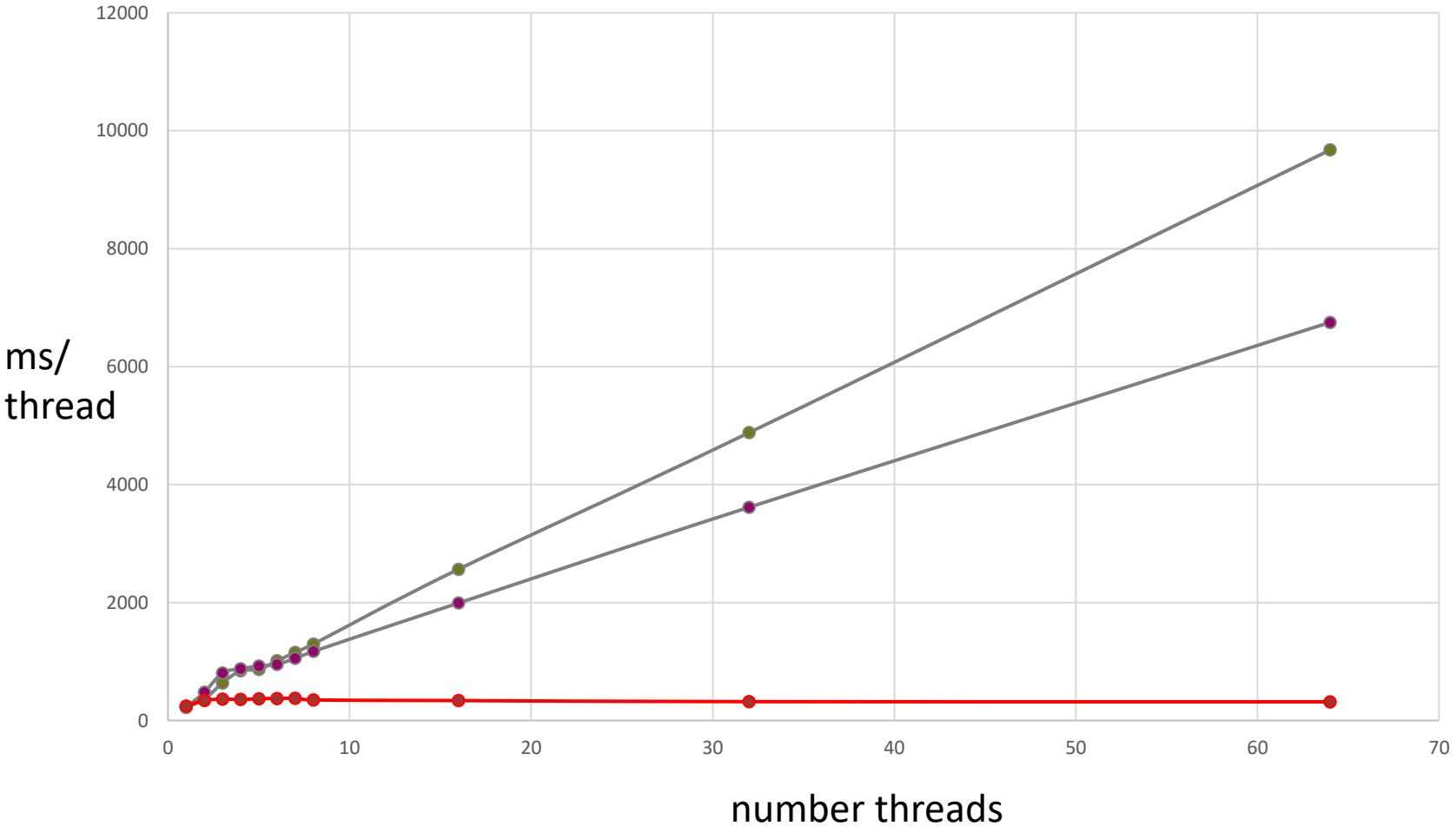
Lock with Backoff

```
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



TAS

TTAS

yeah!

BackoffLock

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?

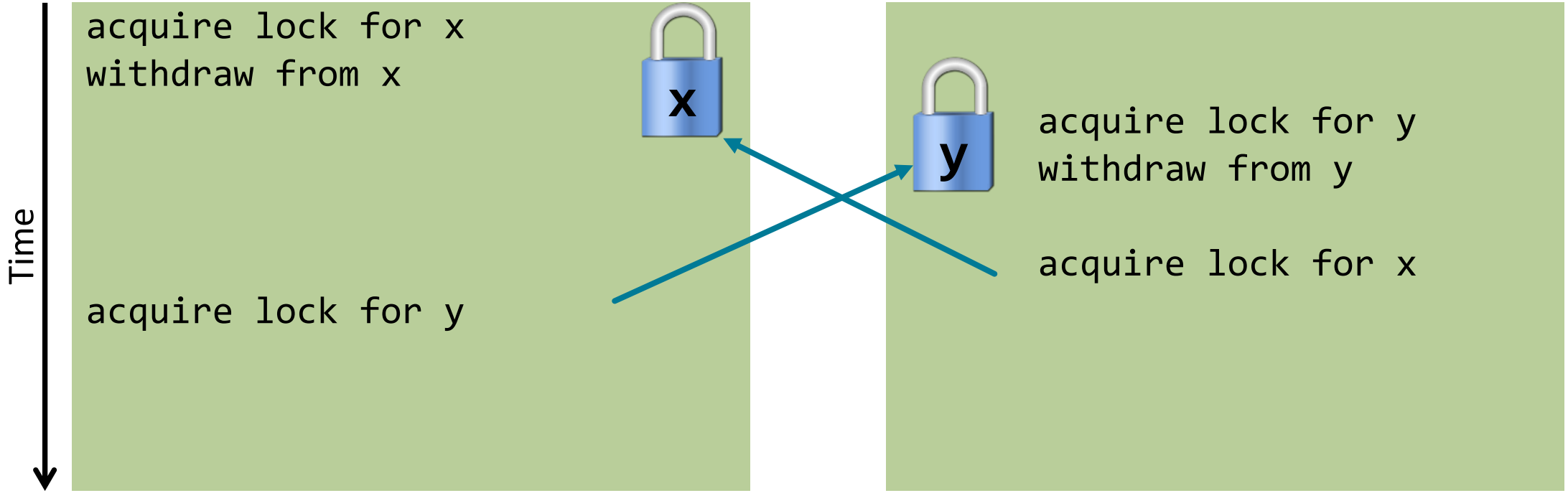
Deadlocks – Motivation

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

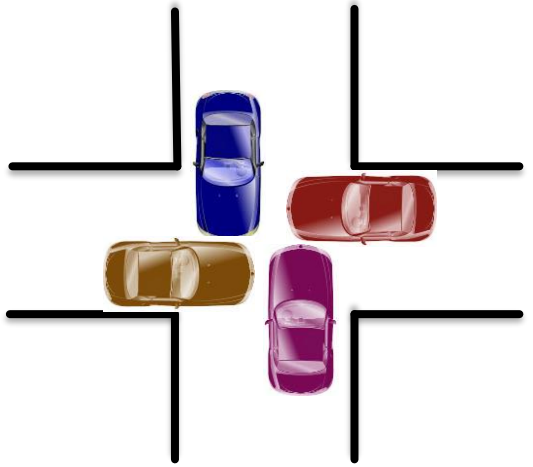
```
class BankAccount {
  …
  synchronized void withdraw(int amount) {…}
  synchronized void deposit(int amount) {…}

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

Thread 1: `x.transferTo(1,y)`

Thread 2: `y.transferTo(1,x)`



Deadlocks

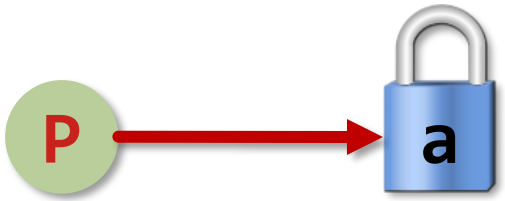


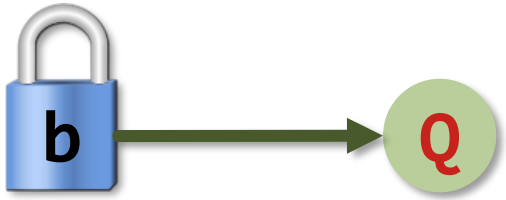
Deadlock: two or more processes are mutually blocked because each process waits for another of these processes to proceed.

Threads and Resources

Graphically: Threads 

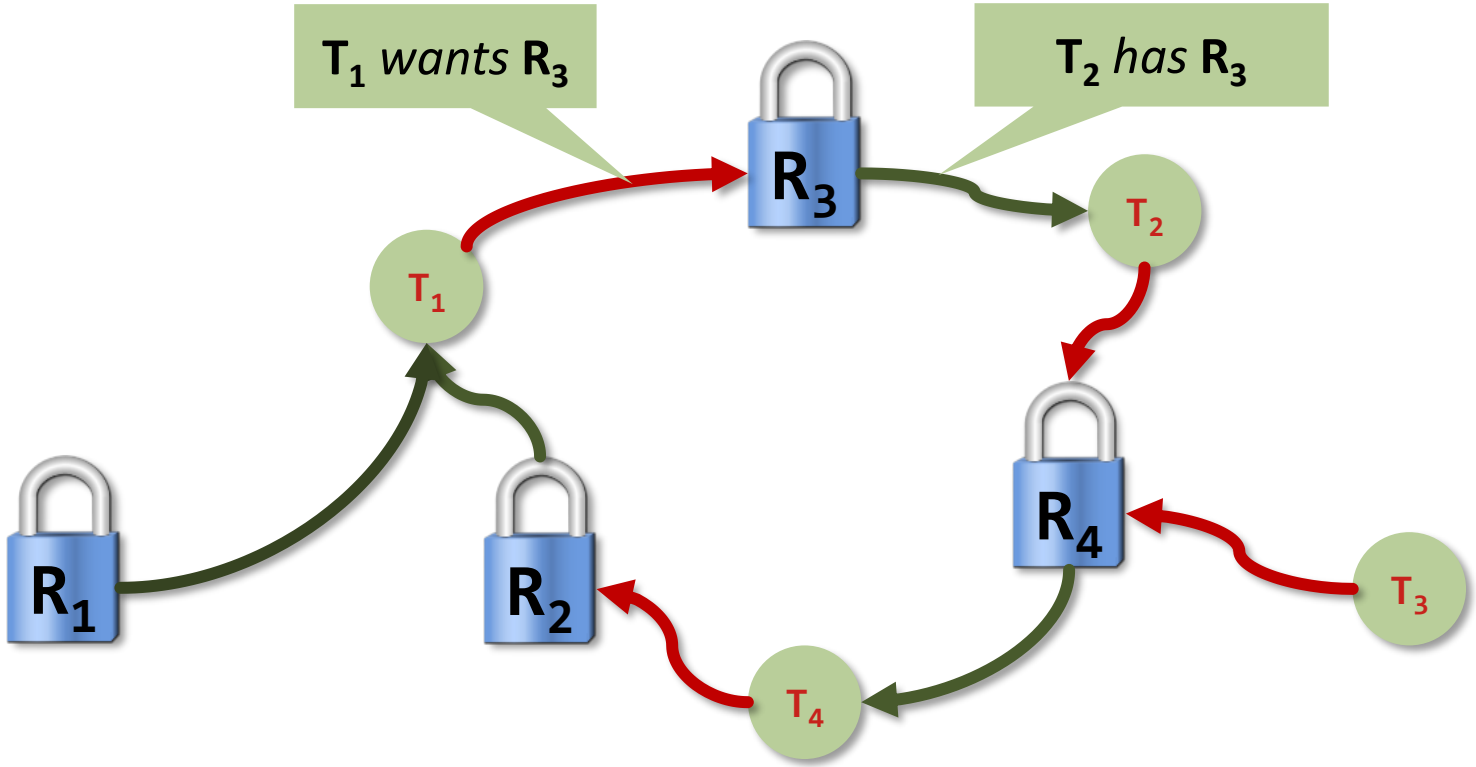
and Resources (Locks) 

Thread P *attempts to* acquire resource a: 

Resource b is *held by* thread Q: 

Deadlocks – more formally

A deadlock for threads $T_1 \dots T_n$ occurs when the directed graph describing the relation of $T_1 \dots T_n$ and resources $R_1 \dots R_m$ 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);
    }
  }
}
```

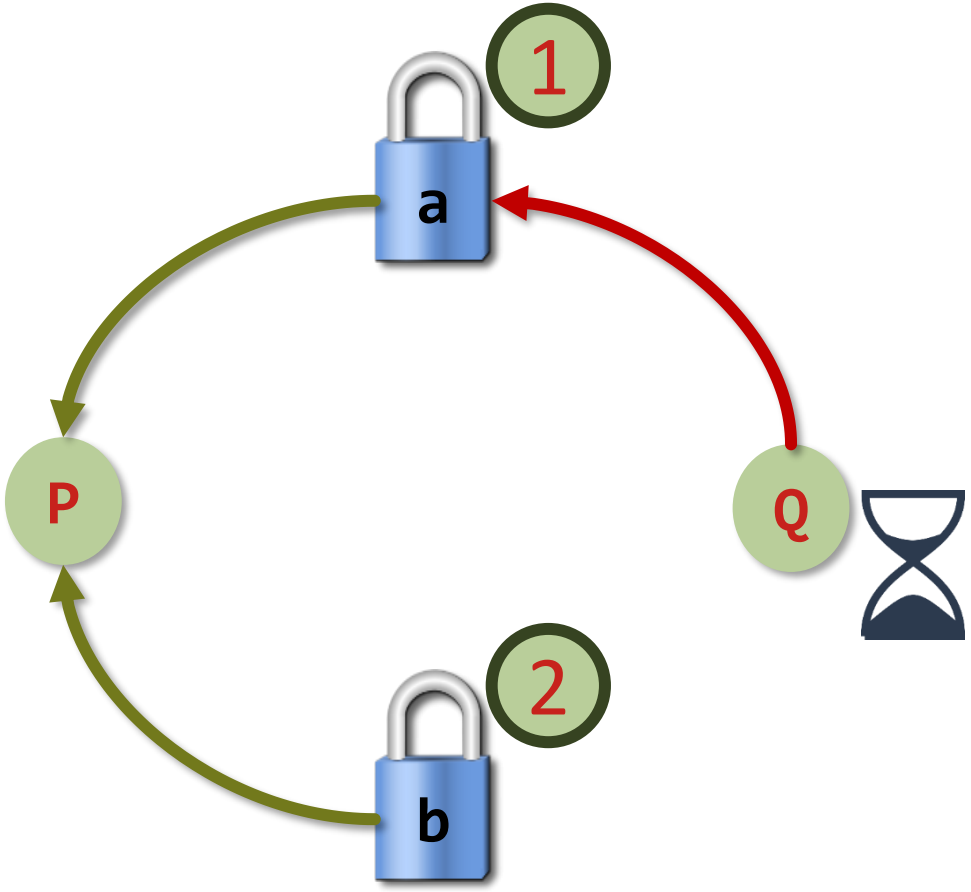
deadlock avoided but no concurrent transfer possible, even not when the pairs of accounts are disjoint.
Often very inefficient!

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 shorther ☺

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 and tricks for programmability

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

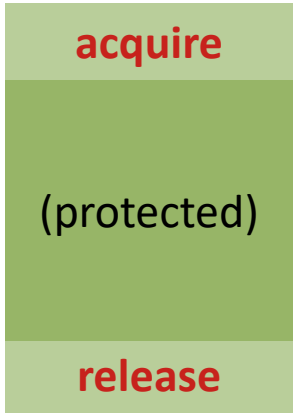
Semaphore: integer-valued abstract data type S with some initial value s≥0 and the following operations*

```
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

When is x ready?

- **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?

```
Thread A
xA=…;
lock();
x=x+xA;
unlock()
```

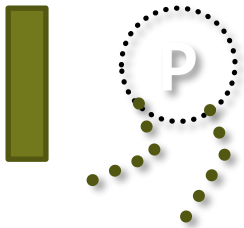
```
Thread B
xB=…;
lock();
x=x+xB;
unlock()
```

```
Thread A
xA=…;
x=x+xA;
release(S);
```

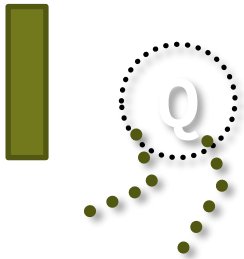
```
Thread A
xB=…;


acquire(S);
x=x+xA;
```

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

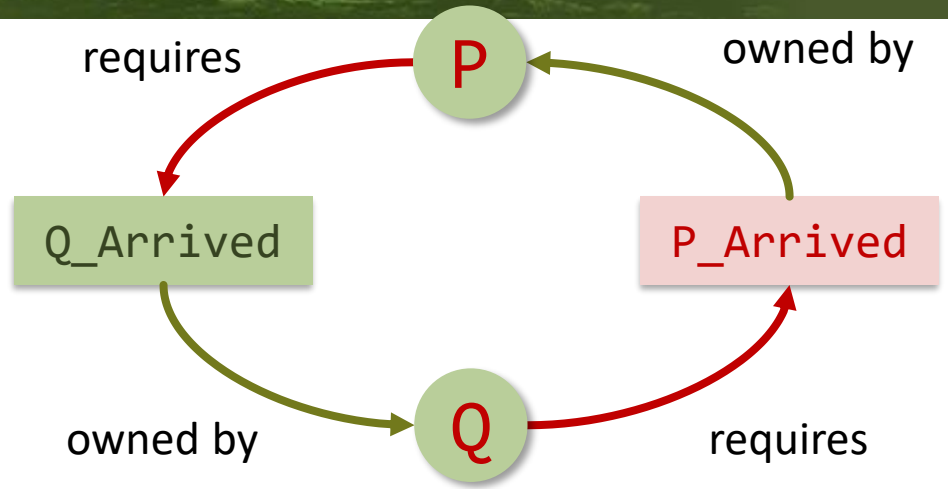
Synchronize Processes P and Q at one location (Rendezvous)

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	`...`	`...`

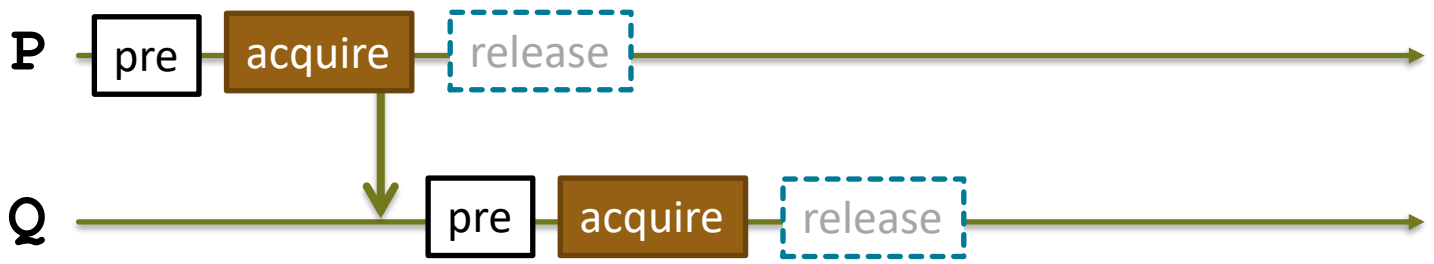
Deadlock



	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	`...`	`..`

Digression: Implementing Semaphores without Spinning (blocking queues)

Consider a process list Q_S associated with semaphore S

acquire(S)

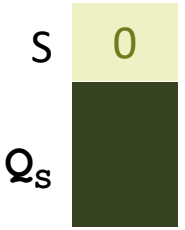
atomic

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

release(S)

atomic

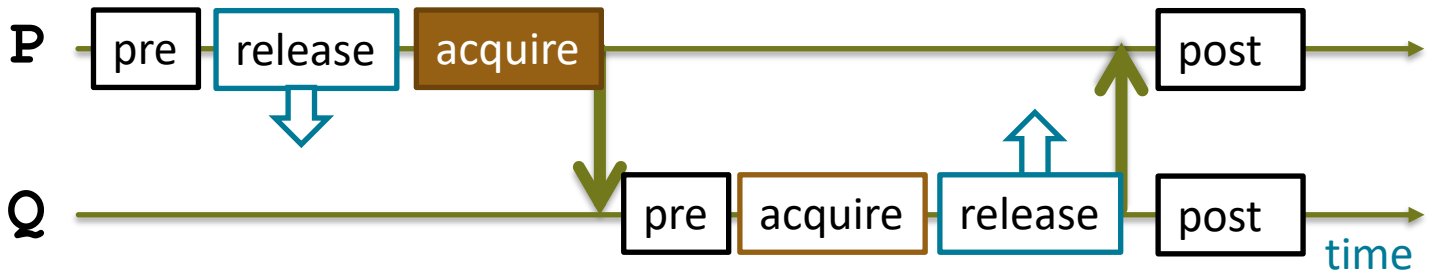
```
{if Qs == ∅ then
    inc(S)
 else
    get(Qs, p)
    unblock(p)
 end }
```



Scheduling Scenarios

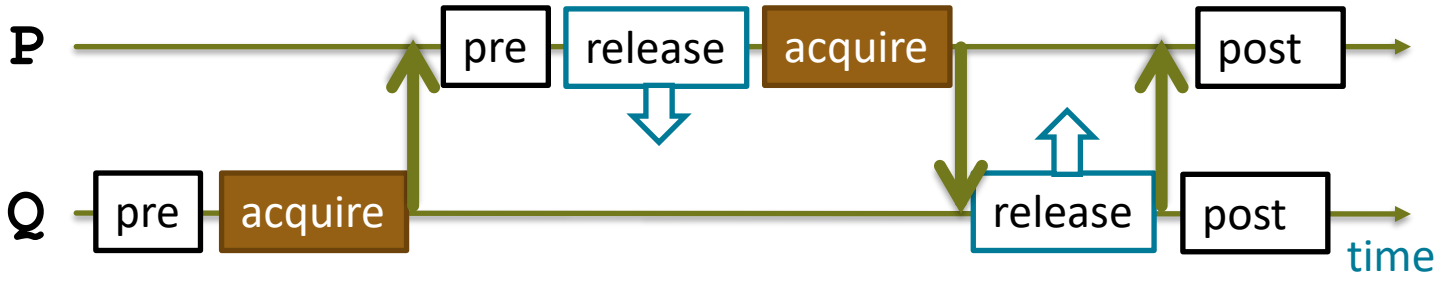
	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `acquire(Q_Arrived)`	`acquire(P_Arrived)` `release(Q_Arrived)`
post	`...`	`..`

P first



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

Q first



Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)

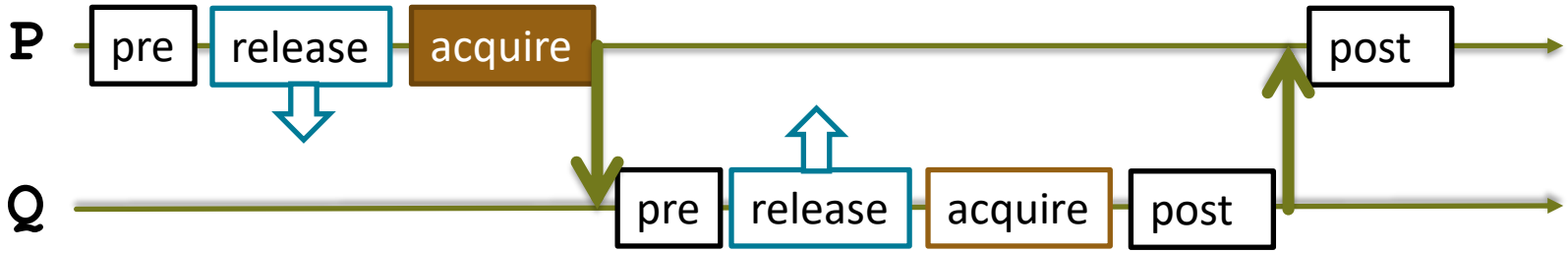
Assume Semaphores `P_Arrived` and `Q_Arrived`

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

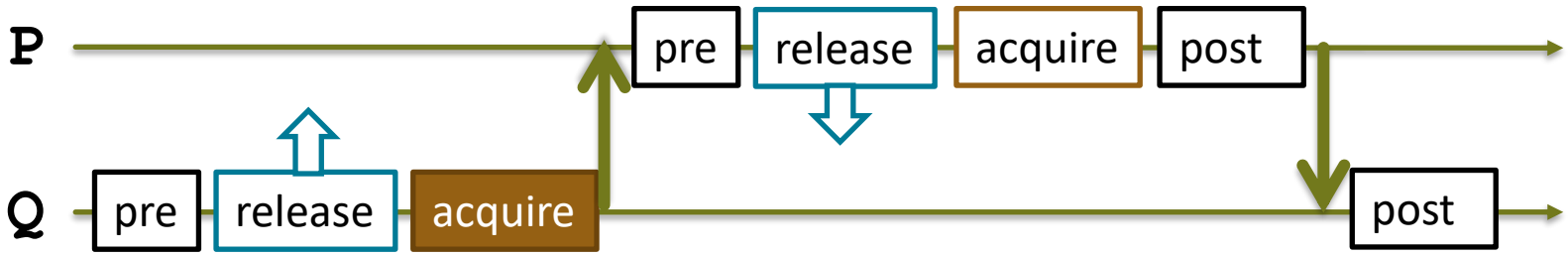
That's even better.

	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! (remember the 57 Pflop/s on 27,360 GPUs on Summit)
  - 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

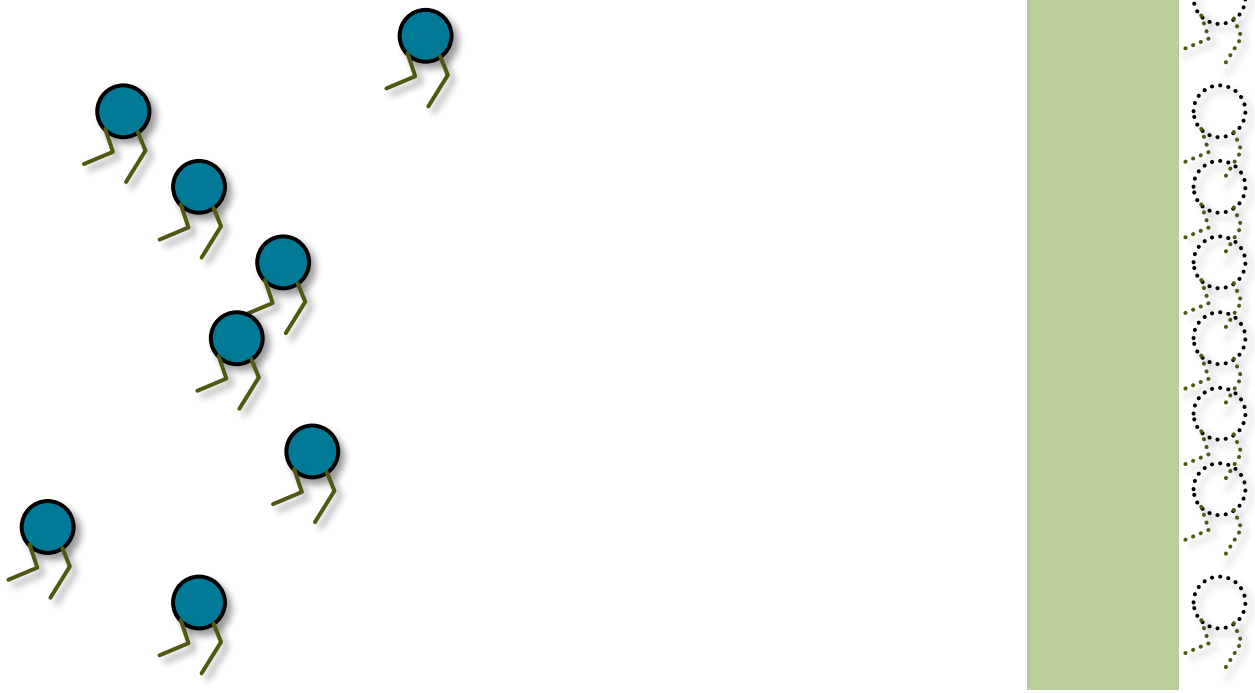
The success of the von Neumann model of sequential computation is attributable to the fact that it is an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware. The author argues that an analogous 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

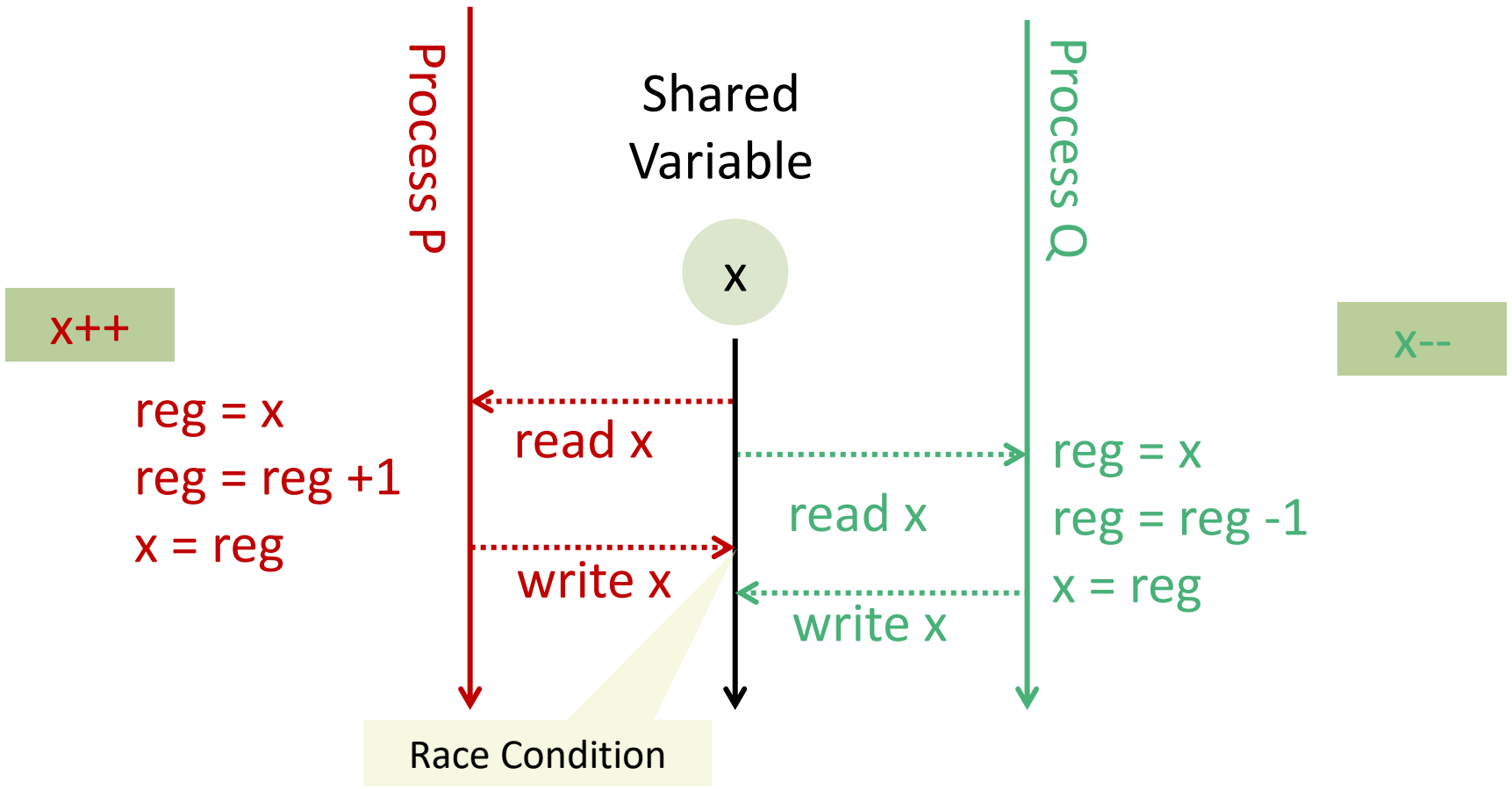
«Each of the processes eventually reaches the acquire statement"

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

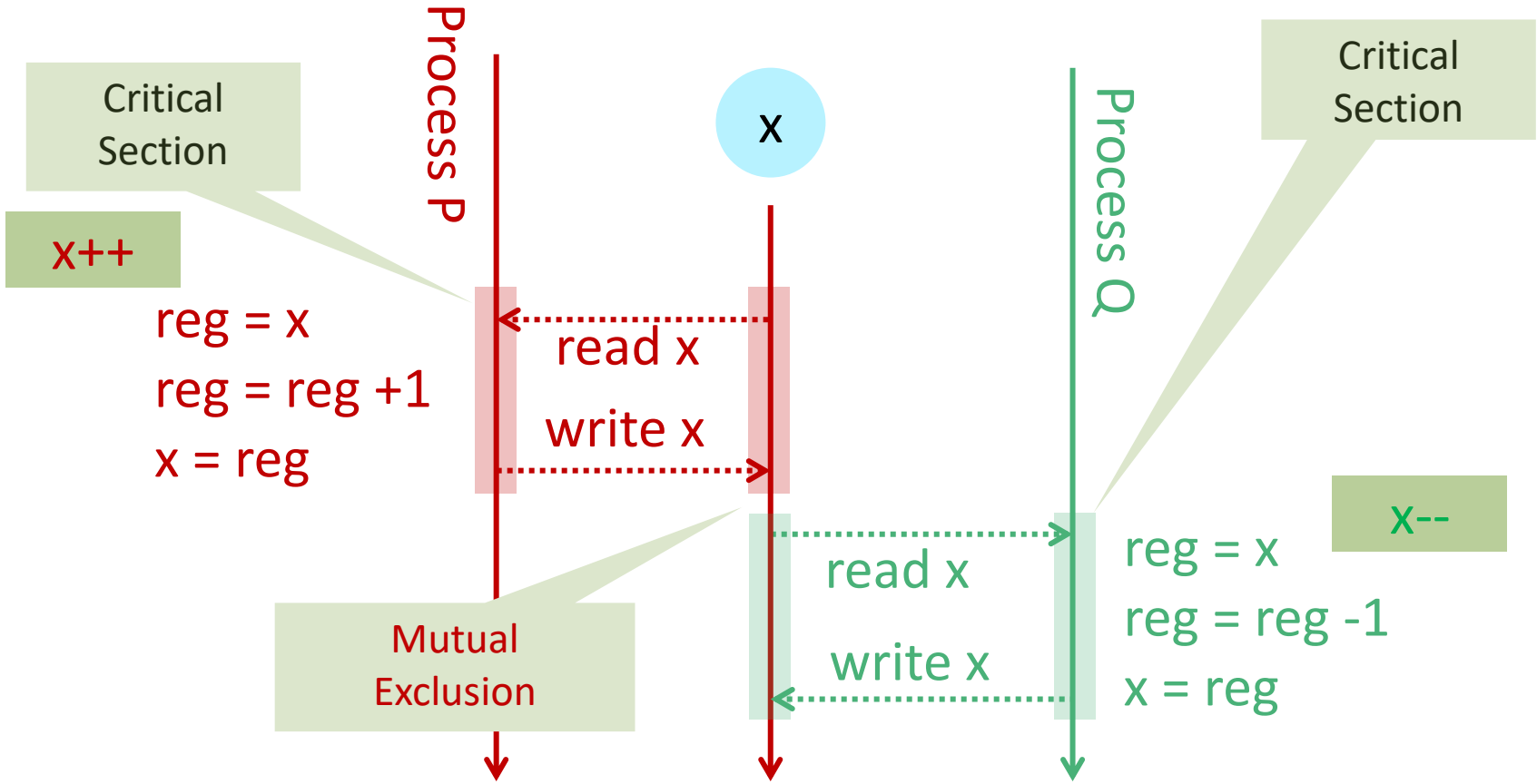
«count provides the number of processes that have passed the barrier" (violated)

«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)`
		←	←	←

post	`...`			

turnstile

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	`...`		

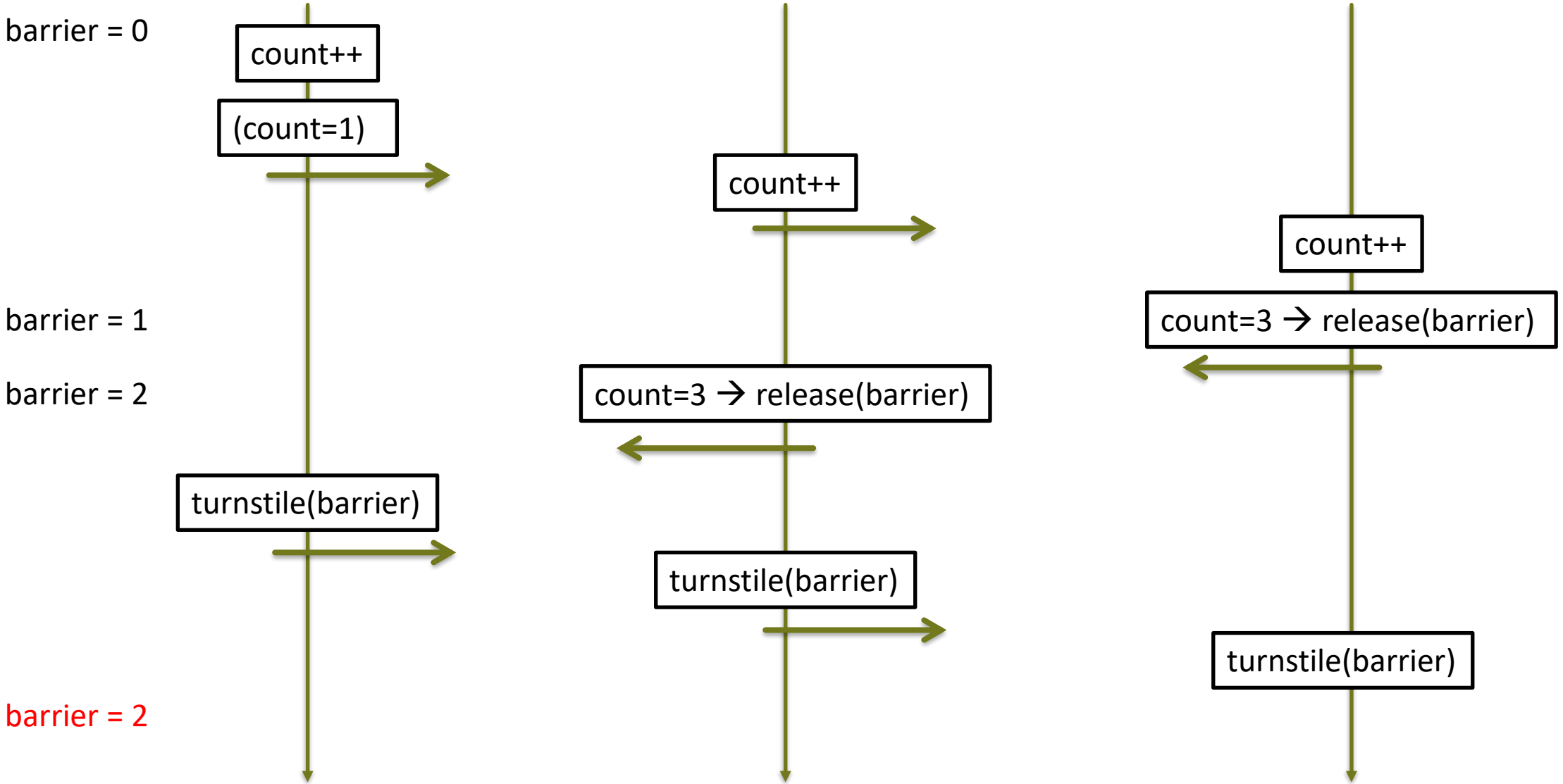
Invariants

«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

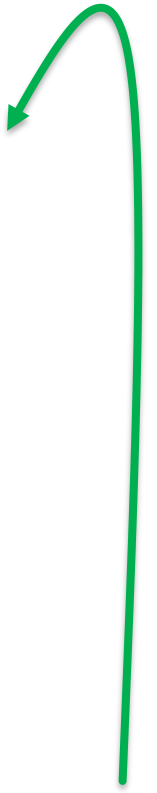


Reusable Barrier. 2nd trial.

	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?

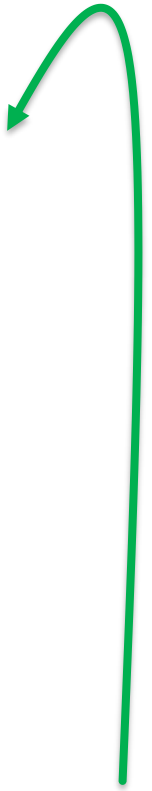
Process can pass other processes!

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

barrier

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

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