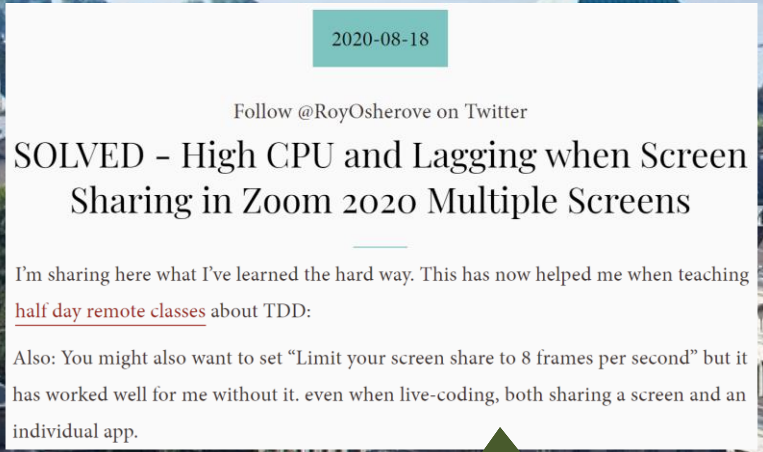
Torsten Hoefler

Parallel Programming

Beyond Locks II: Semaphore, Barrier, Producer-/Consumer, Monitors



2020-08-18

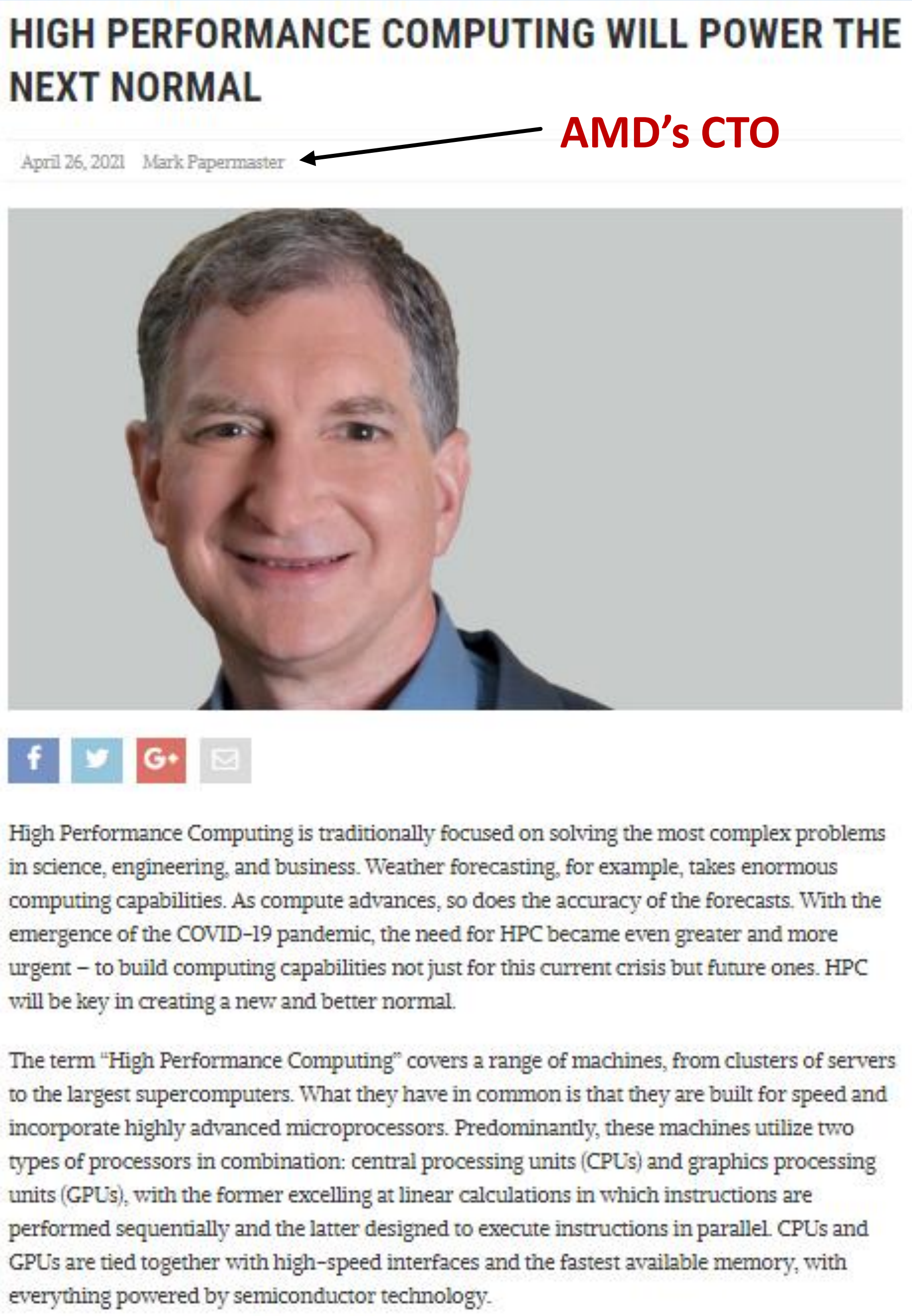
Follow @RoyOsherove on Twitter

SOLVED – High CPU and Lagging when Screen Sharing in Zoom 2020 Multiple Screens

I'm sharing here what I've learned the hard way. This has now helped me when teaching half day remote classes about TDD:

Also: You might also want to set "Limit your screen share to 8 frames per second" but it has worked well for me without it. even when live-coding, both sharing a screen and an individual app.

Nice example for priority problems in a pipeline!



HIGH PERFORMANCE COMPUTING WILL POWER THE NEXT NORMAL

April 26, 2021 Mark Papermaster

AMD's CTO

High Performance Computing is traditionally focused on solving the most complex problems in science, engineering, and business. Weather forecasting, for example, takes enormous computing capabilities. As compute advances, so does the accuracy of the forecasts. With the emergence of the COVID-19 pandemic, the need for HPC became even greater and more urgent – to build computing capabilities not just for this current crisis but future ones. HPC will be key in creating a new and better normal.

The term "High Performance Computing" covers a range of machines, from clusters of servers to the largest supercomputers. What they have in common is that they are built for speed and incorporate highly advanced microprocessors. Predominantly, these machines utilize two types of processors in combination: central processing units (CPUs) and graphics processing units (GPUs), with the former excelling at linear calculations in which instructions are performed sequentially and the latter designed to execute instructions in parallel. CPUs and GPUs are tied together with high-speed interfaces and the fastest available memory, with everything powered by semiconductor technology.

Learning goals for today

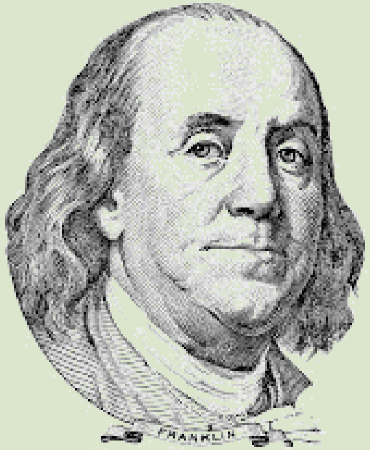
- **So far:**
  - Proof of correctness of parallel programs (example locks)
  - Proof of starvation freedom
  - Multi-thread locks using (atomic and weaker) registers (memory)

- **Now:**
  - Atomic operations – more realistic locks with Read-Modify-Write operations
  - Concurrency on a higher level: Deadlocks, Semaphores, Barriers

- **Learning goals:**
  - Understand atomic operations – first impressions for now
  - More advanced synchronization operations

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."

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 <span>Read-Modify-Write</span> («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);
```

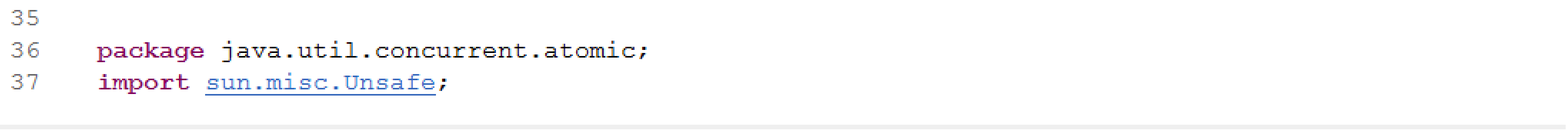
atomically set to value update iff current value is expect. Return true on success.

sets newValue and returns previous value.

How does this work? (for experts)

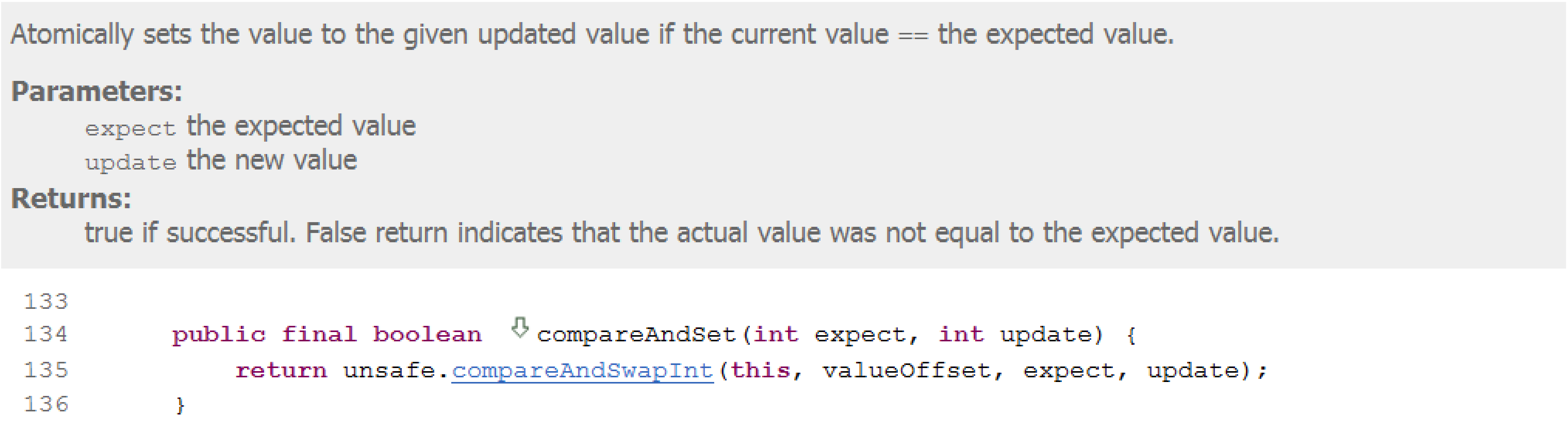
- **The JVM bytecode does not offer atomic operations like CAS. [It does, however, support monitors via instructions monitorenter, monitorexit, we will understand this later]**
- **But there is a (yet undocumented) class `sun.misc.Unsafe` offering direct mappings from java to underlying machine / OS.**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

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



```

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.

```

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

(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);
    }
    ...
}
```

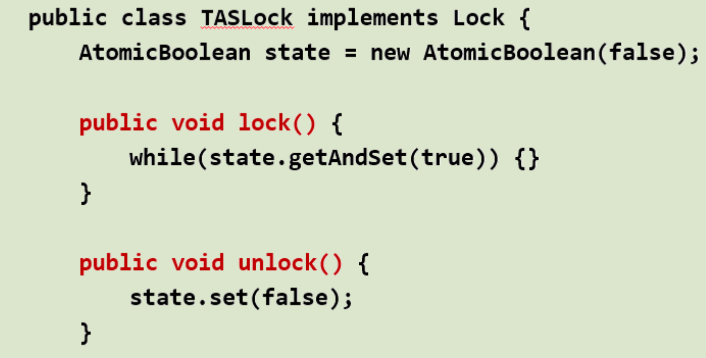
Spinlock:

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

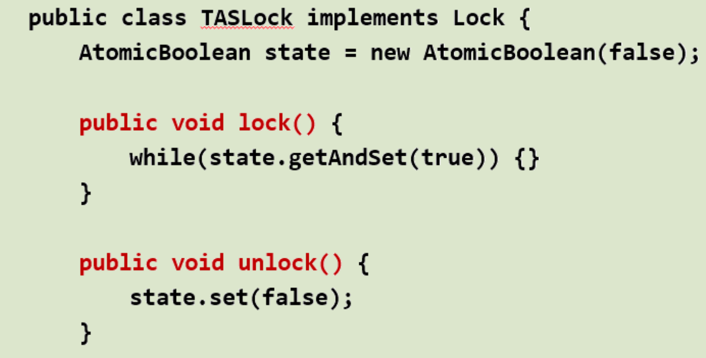
Intel core i7@3.4 GHz, 4 cores + HT

Why?

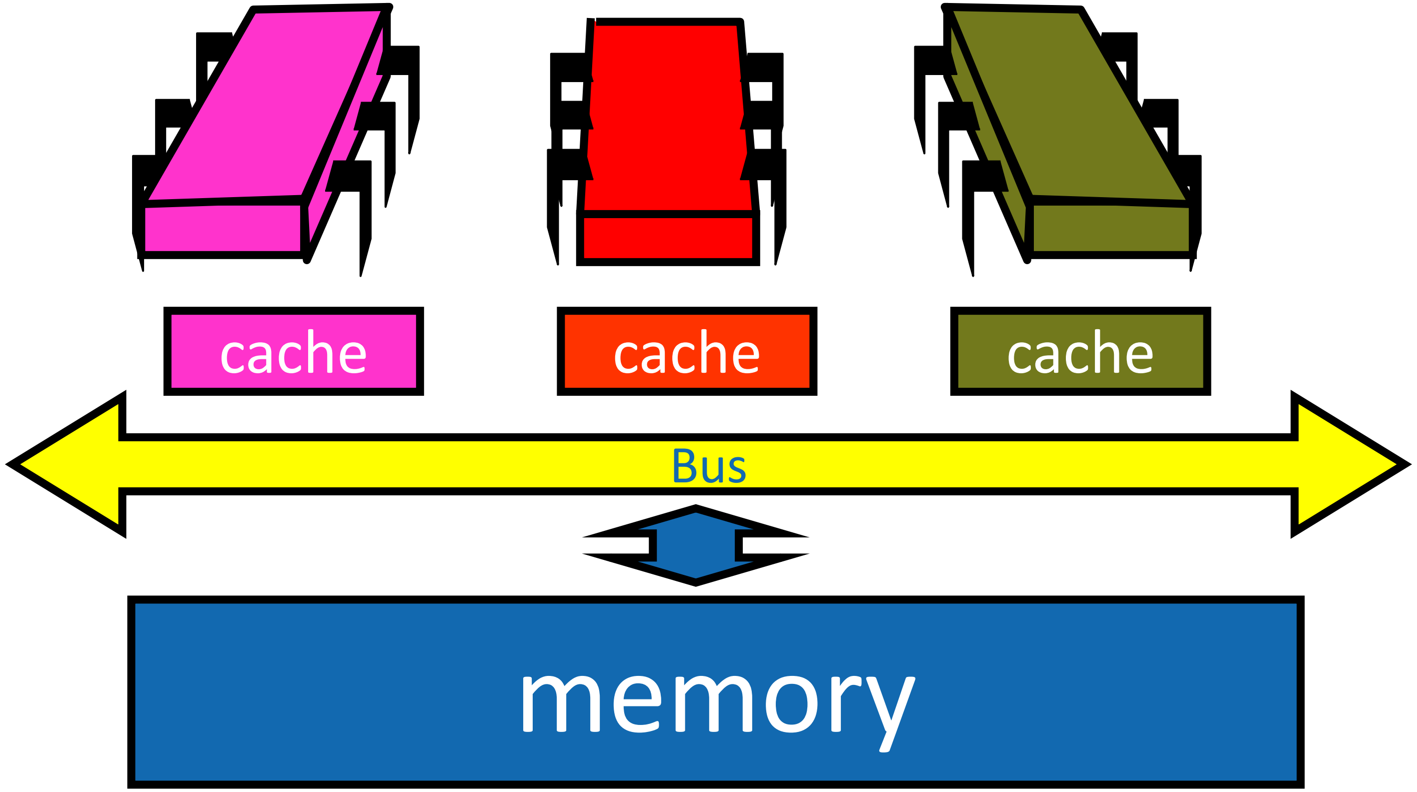
sequential bottleneck

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

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



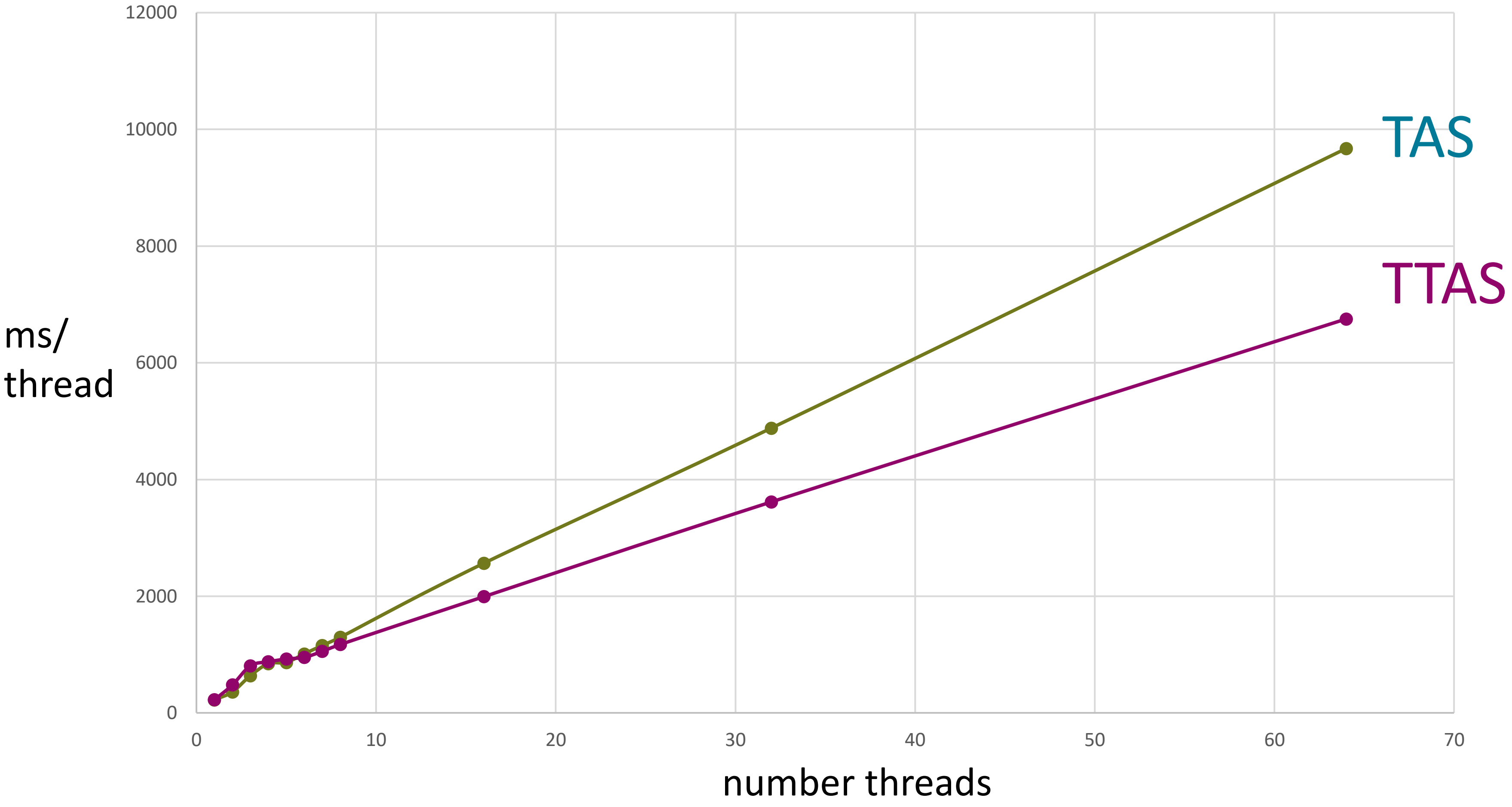
```
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

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

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



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);
}
```

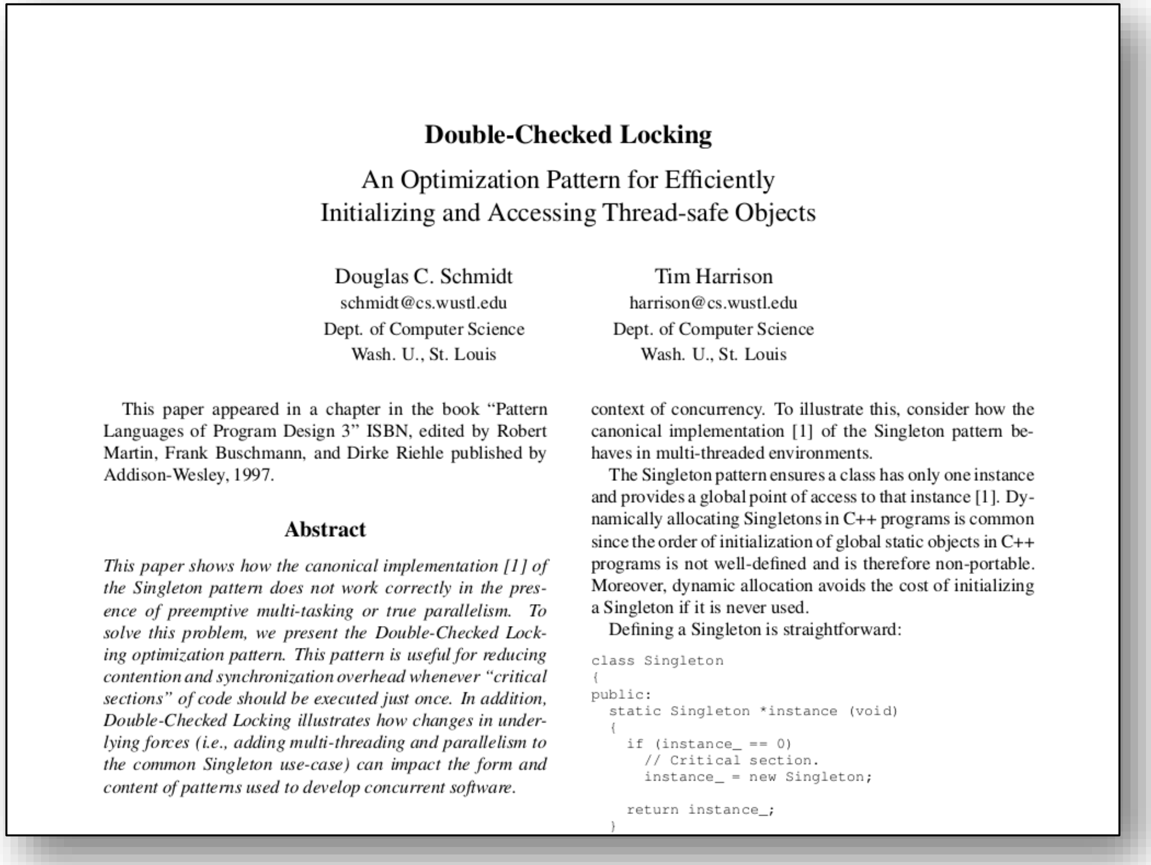
Measurement



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

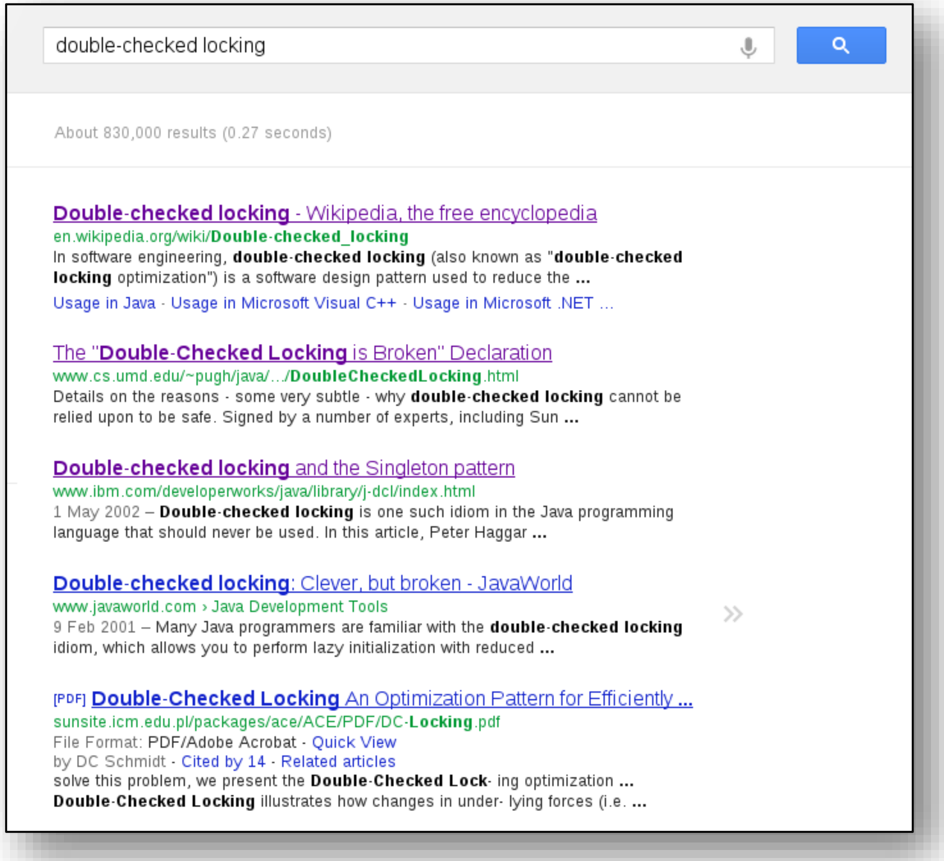
This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and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

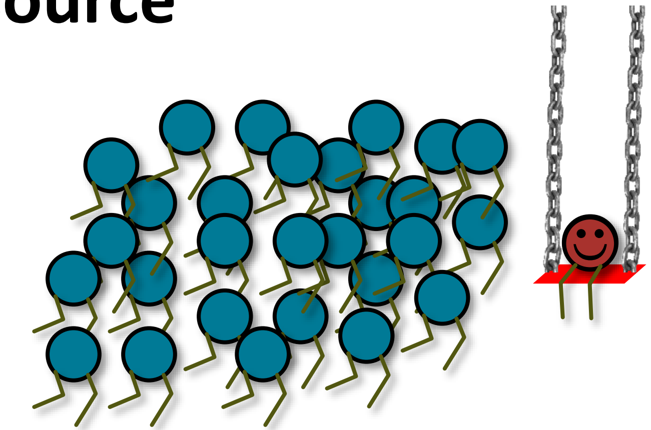
[PDF] **Double-Checked Locking** An Optimization Pattern for Efficiently ...
sunsite.icm.edu.pl/packages/ace/ACE/PDF/DC-**Locking**.pdf
File Format: PDF/Adobe Acrobat - Quick View
by DC Schmidt - Cited by 14 - Related articles
solve this problem, we present the **Double-Checked Lock**- ing optimization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

Problem: Memory ordering leads to race-conditions!

TATAS with backoff

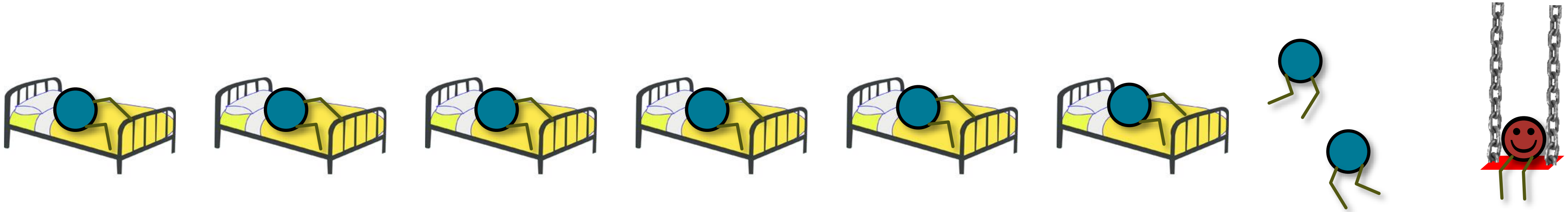
Observation

- (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



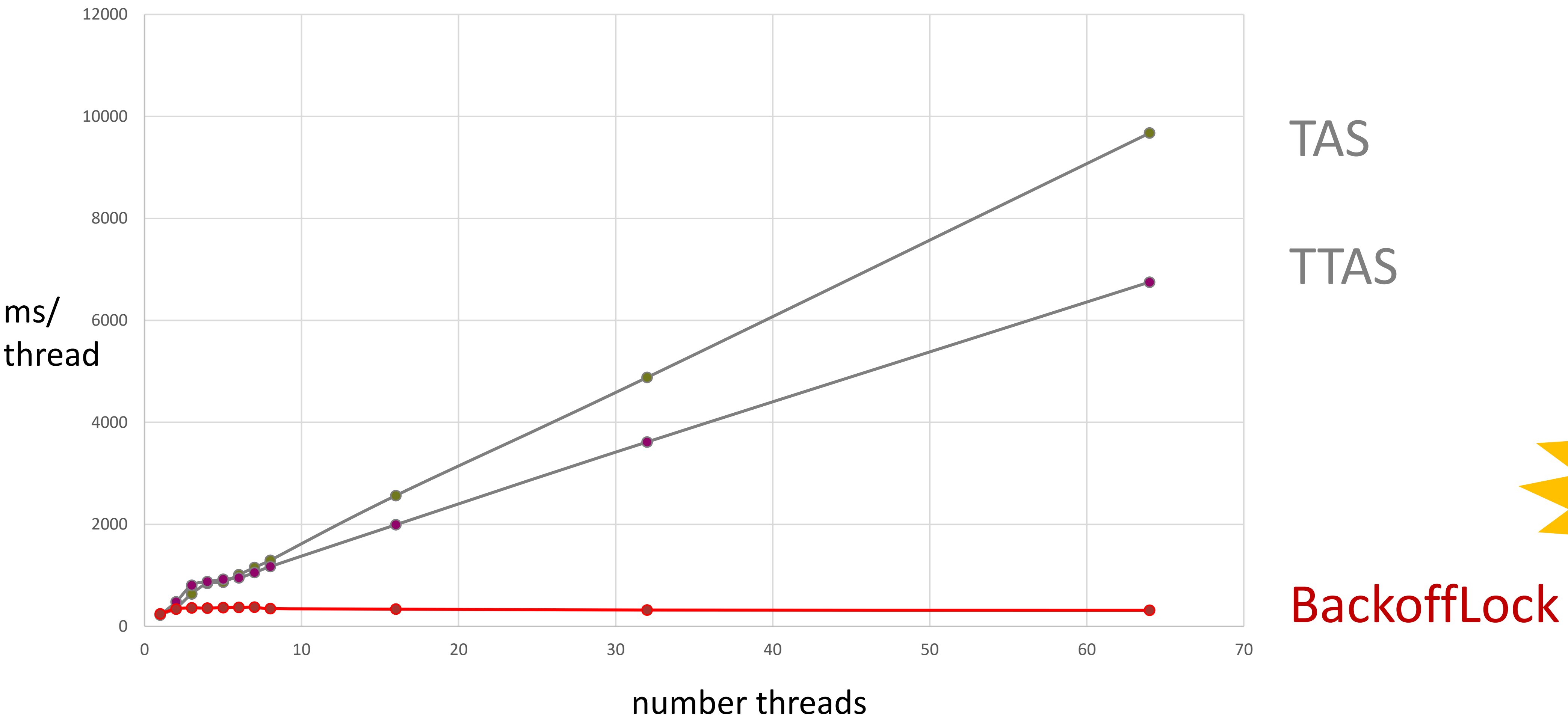
Lock with Backoff

```
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



yeah!

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?

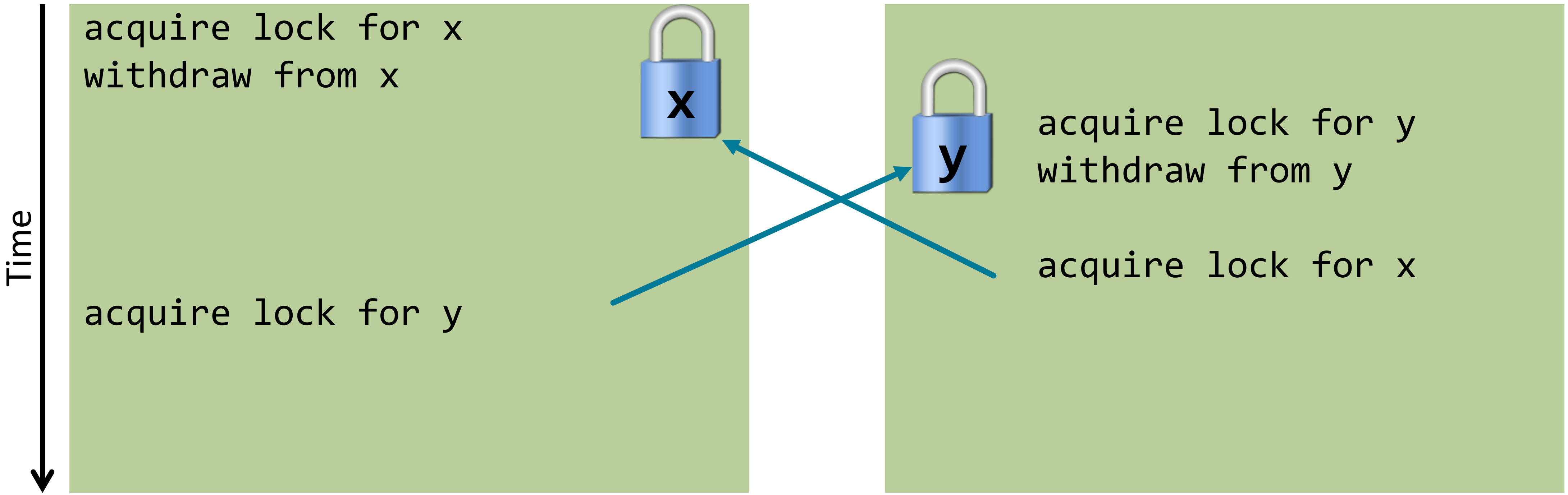
Deadlocks – Motivation

```
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

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

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



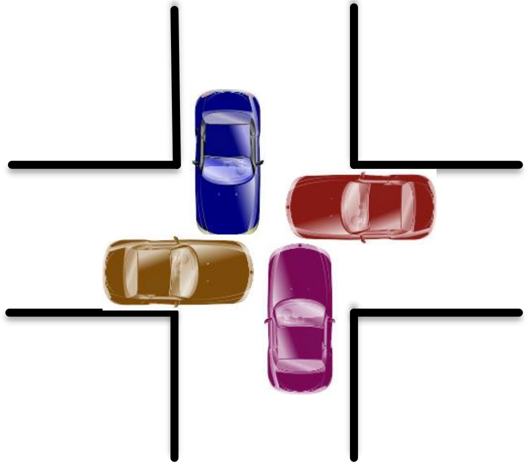
Thread 1:

```
acquire lock for x
withdraw from x

acquire lock for y
```

Thread 2:

```
acquire lock for y
withdraw from y

acquire lock for x
```

Deadlocks

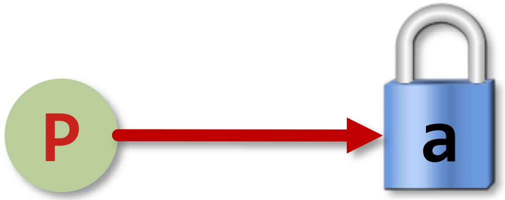


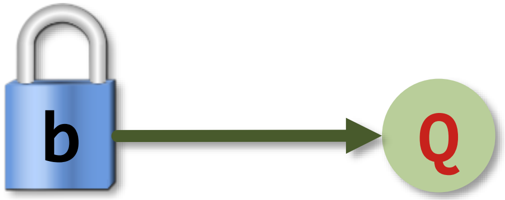
Deadlock: two or more processes are mutually blocked because each process waits for another of these processes to proceed.

Threads and Resources

Graphically: Threads

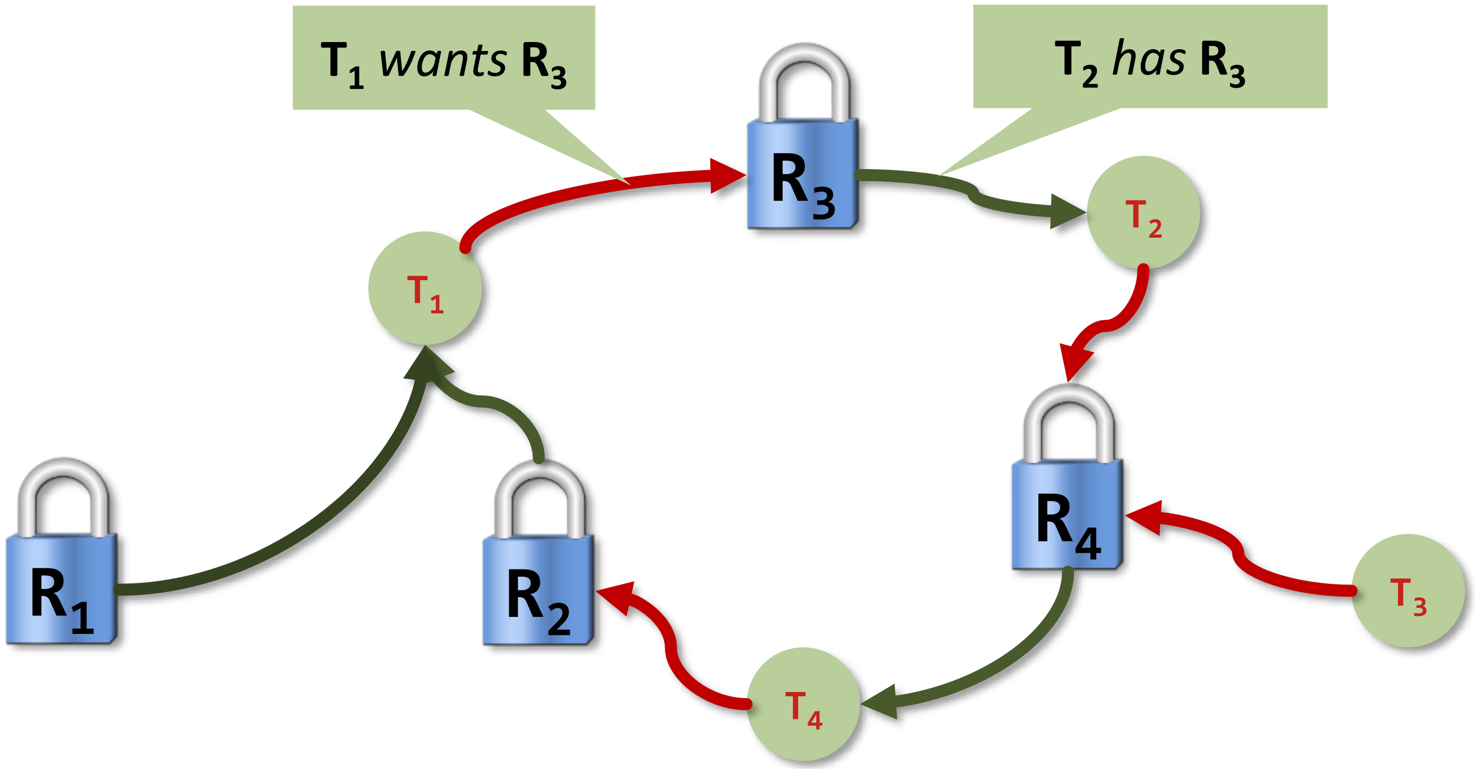
and Resources (Locks)

Thread P *attempts to* acquire resource a: 

Resource b is *held by* thread Q: 

Deadlocks – more formally

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


T$_1$ *wants* **R$_3$**

T$_2$ *has* **R$_3$**

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);
      }
   }
}
```

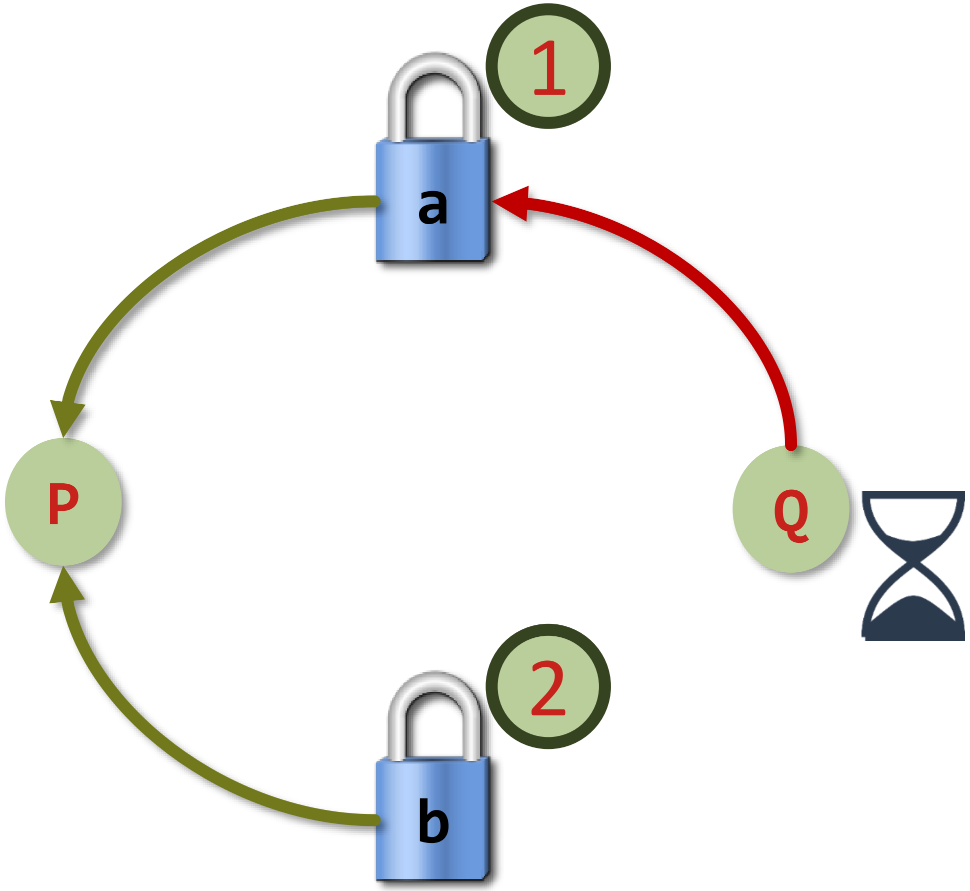
deadlock avoided but no concurrent transfer possible, even not when the pairs of accounts are disjoint.
Often very inefficient!

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

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

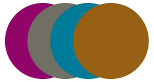
Semaphore: integer-valued abstract data type `S` with some initial value `s`≥0 and the following operations*

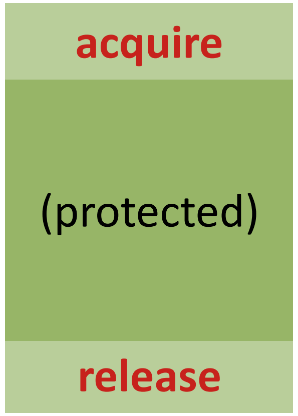
```
acquire(S)
 {
   wait until S > 0
   dec(S)
 }
```

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

sem_mutex = Semaphore(1);

lock mutex := sem_mutex.acquire()

only one thread is allowed into the critical section

unlock mutex := sem_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?

When is x ready?

Thread A

x_A=…;
lock();
x=x+x_A;
unlock()

Thread B

x_B=…;
lock();
x=x+x_B;
unlock()

Thread A

x_A=…;
x=x+x_A;
release(S);

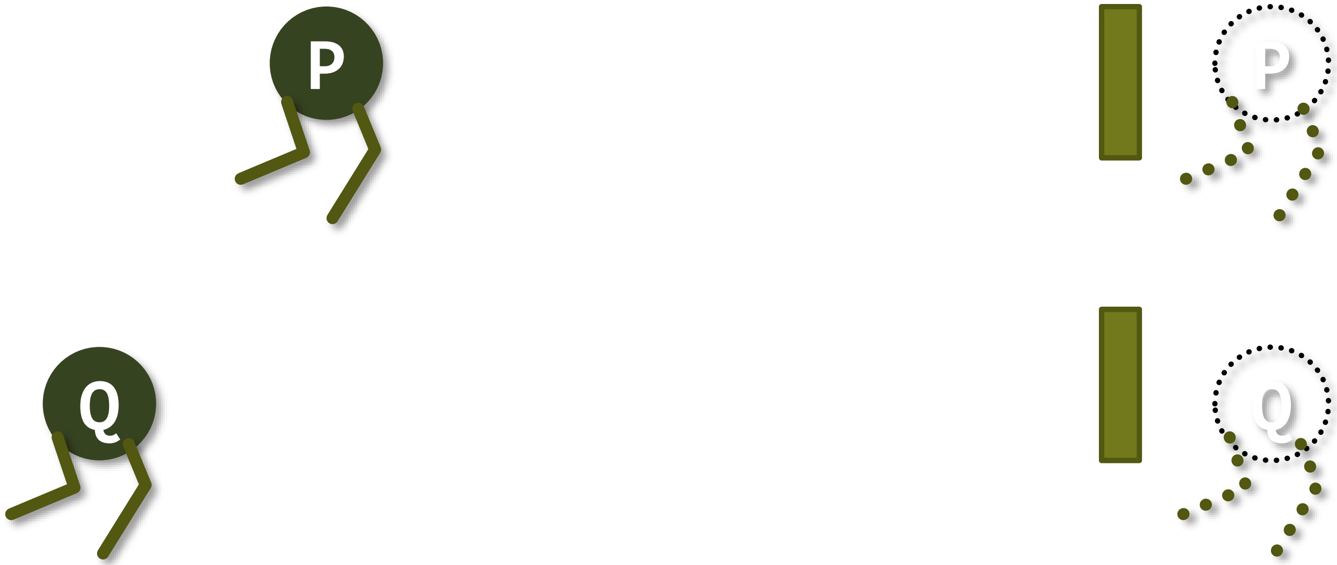
Thread A

x_B=…;

acquire(S);
x=x+x_A;

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

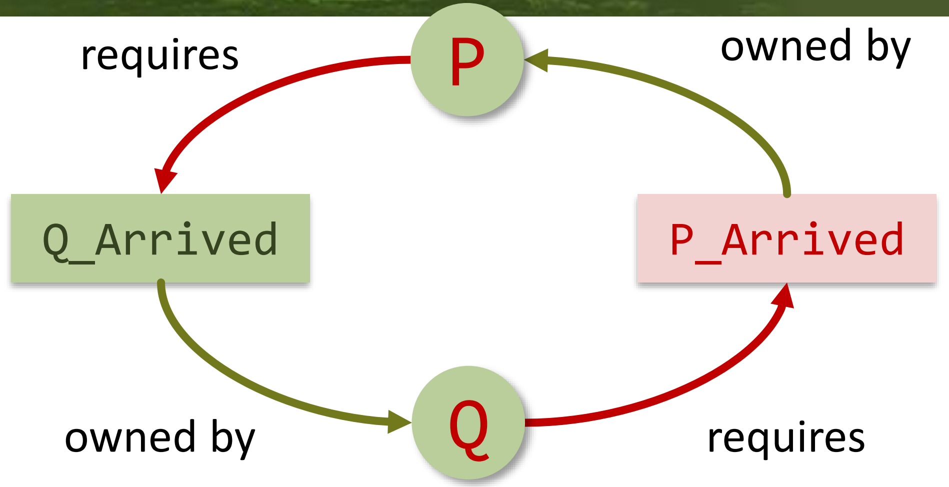
Dou you find the problem?

	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



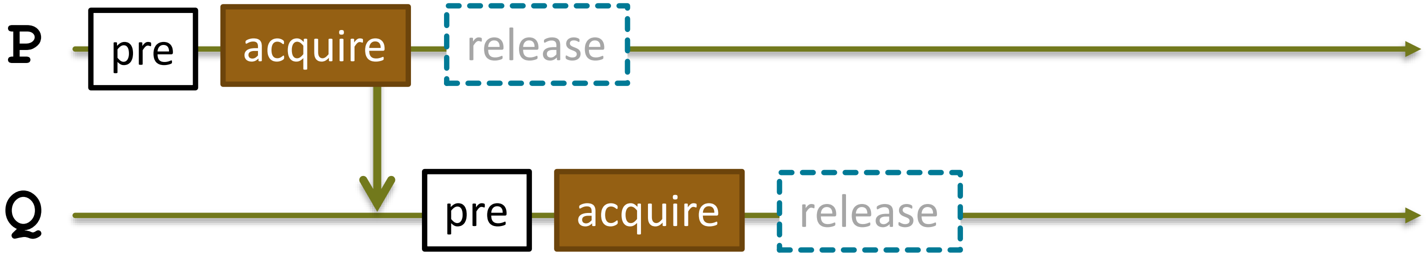
	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`

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)`

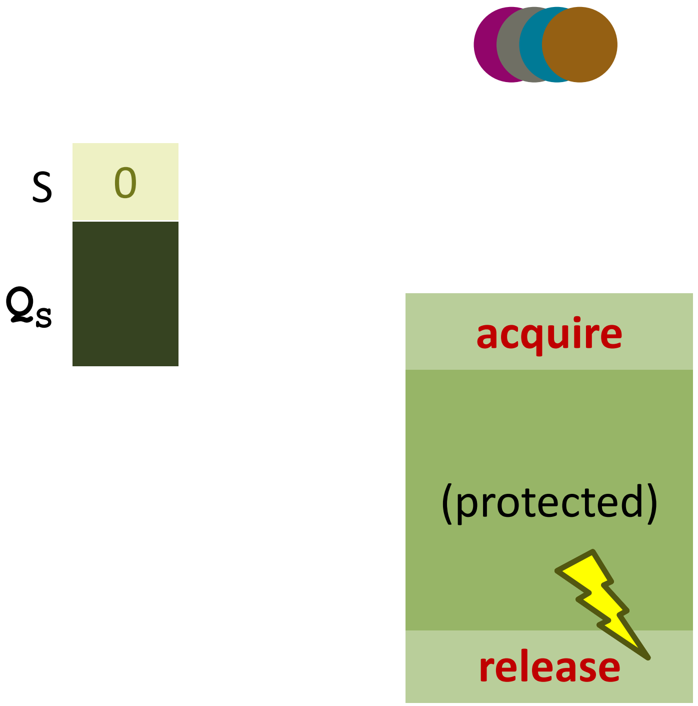
post	`...`	`..`

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(Qs, self)
     block(self)
 end }
```

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



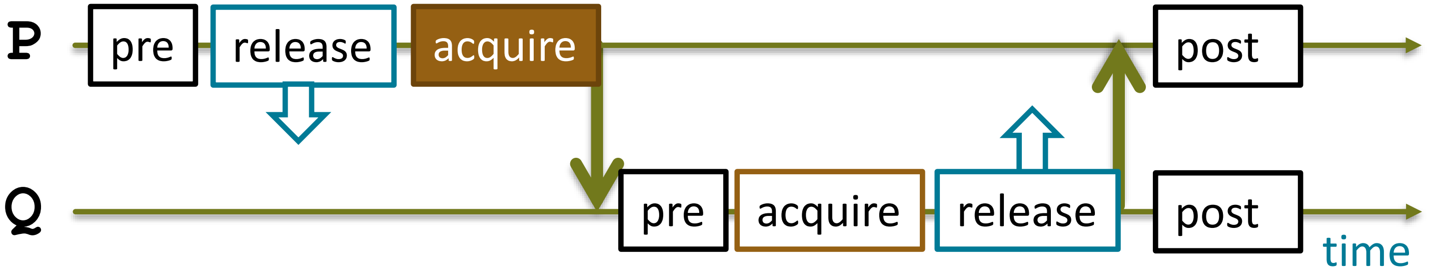
Scheduling Scenarios

	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`

rendezvous	`release(P_Arrived)`
	`acquire(Q_Arrived)`
		`acquire(P_Arrived)`
		`release(Q_Arrived)`

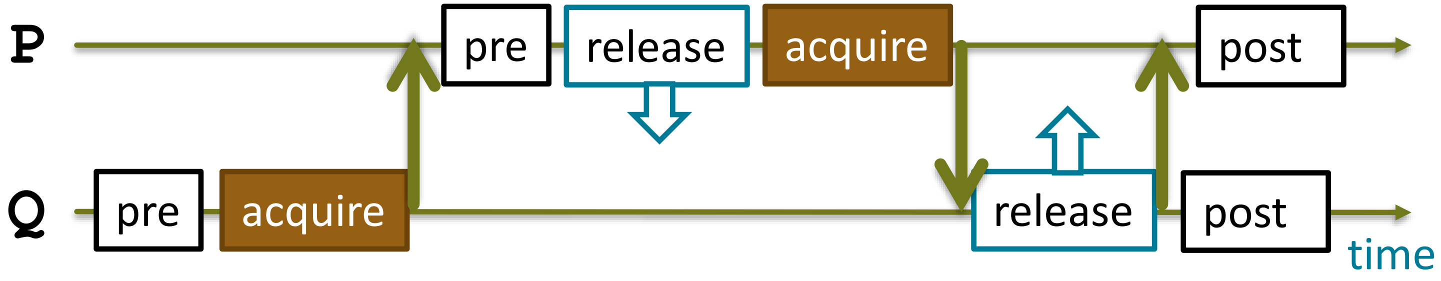
post	`...`	`..`

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	`...`	`..`

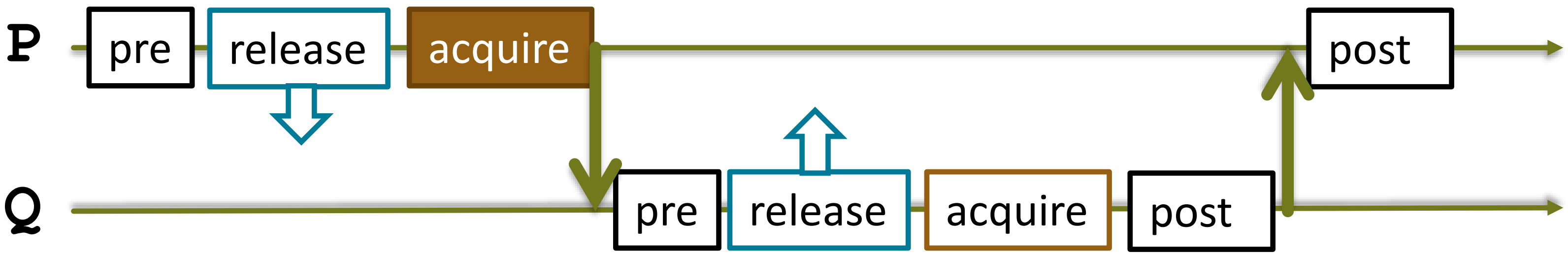
That’s even better.

	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`

rendezvous	`release(P_Arrived)`
	`acquire(Q_Arrived)`
		`release(Q_Arrived)`
		`acquire(P_Arrived)`

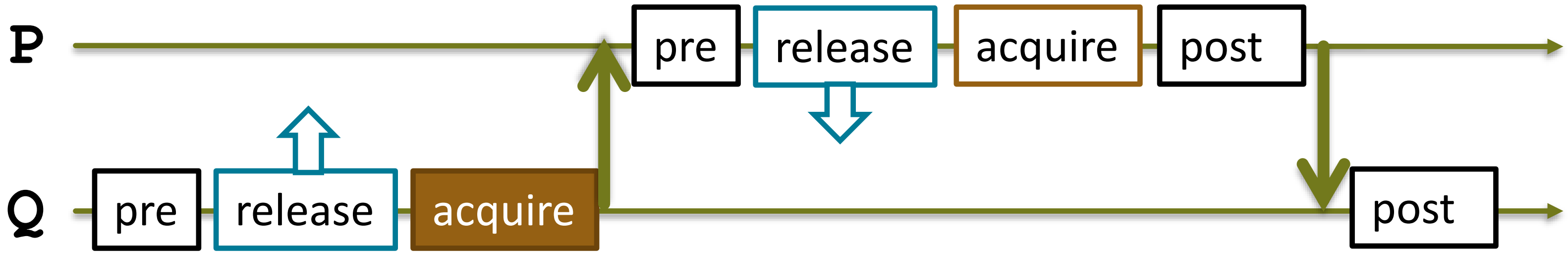
post	`...`	`..`

P first



Q first

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



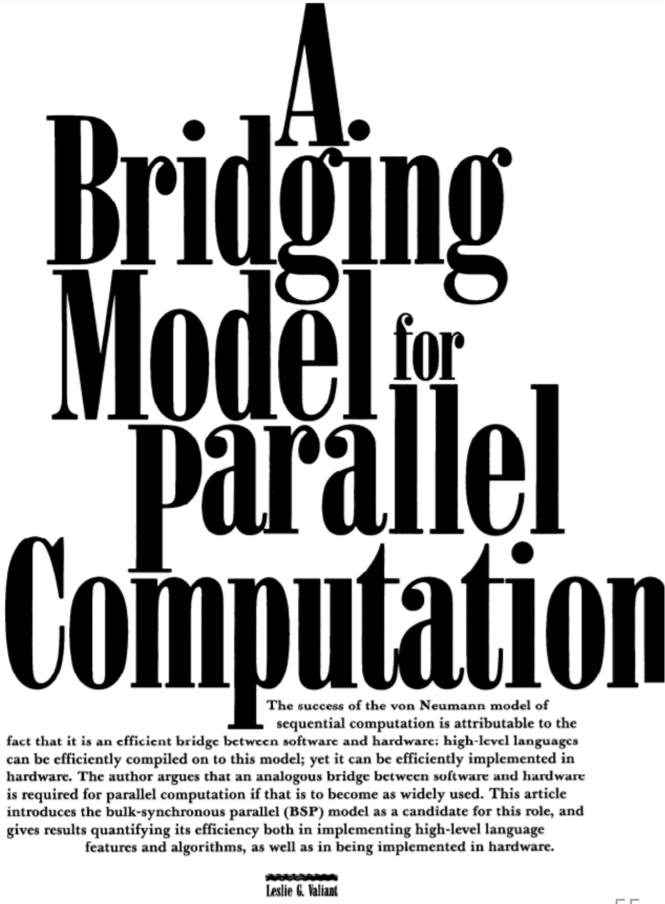
Back to our dot-product

- **Assume now vectors with 1 million entries on 10,000 threads**
  - Very common! (we regularly run >1M threads on 20k+ 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

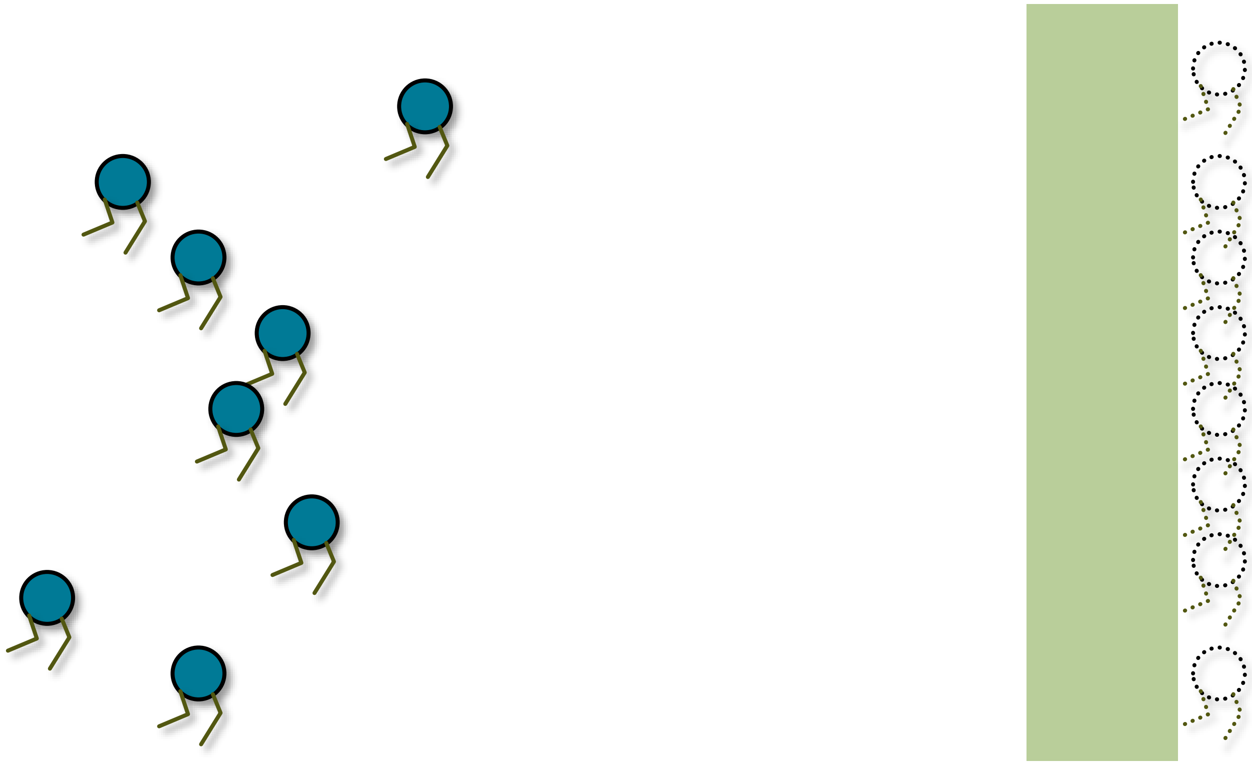
The success of the von Neumann model of sequential computation is attributable to the fact that it is an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware. The author argues that an analogous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

Race Condition !

WRONG

Some wait forever!

Barrier – 1st try

Synchronize a number (n) of processes.

Semaphore barrier. Integer count.

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

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

post	`...`

WRONG

Invariants

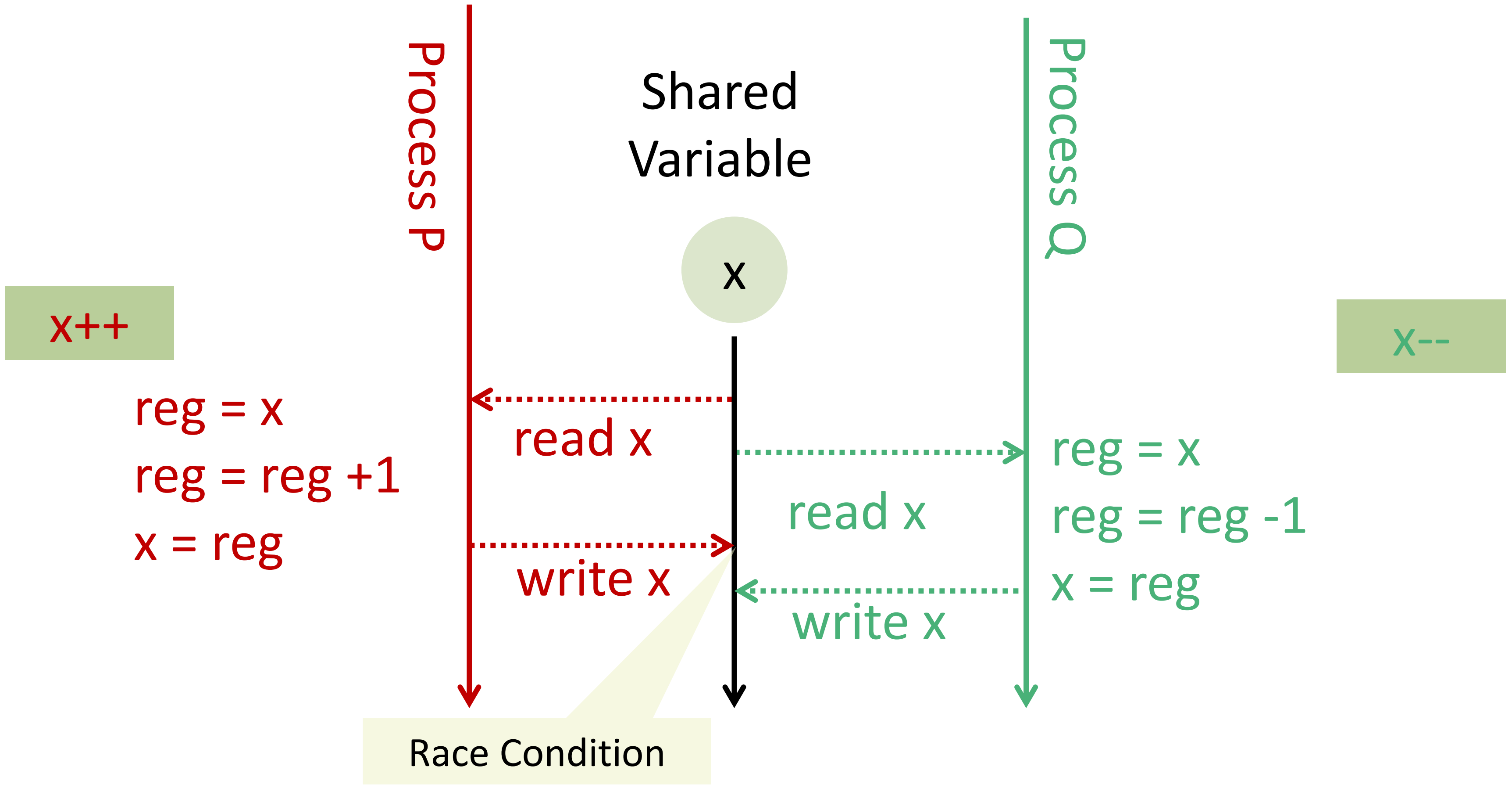
«Each of the processes eventually reaches the acquire statement"

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

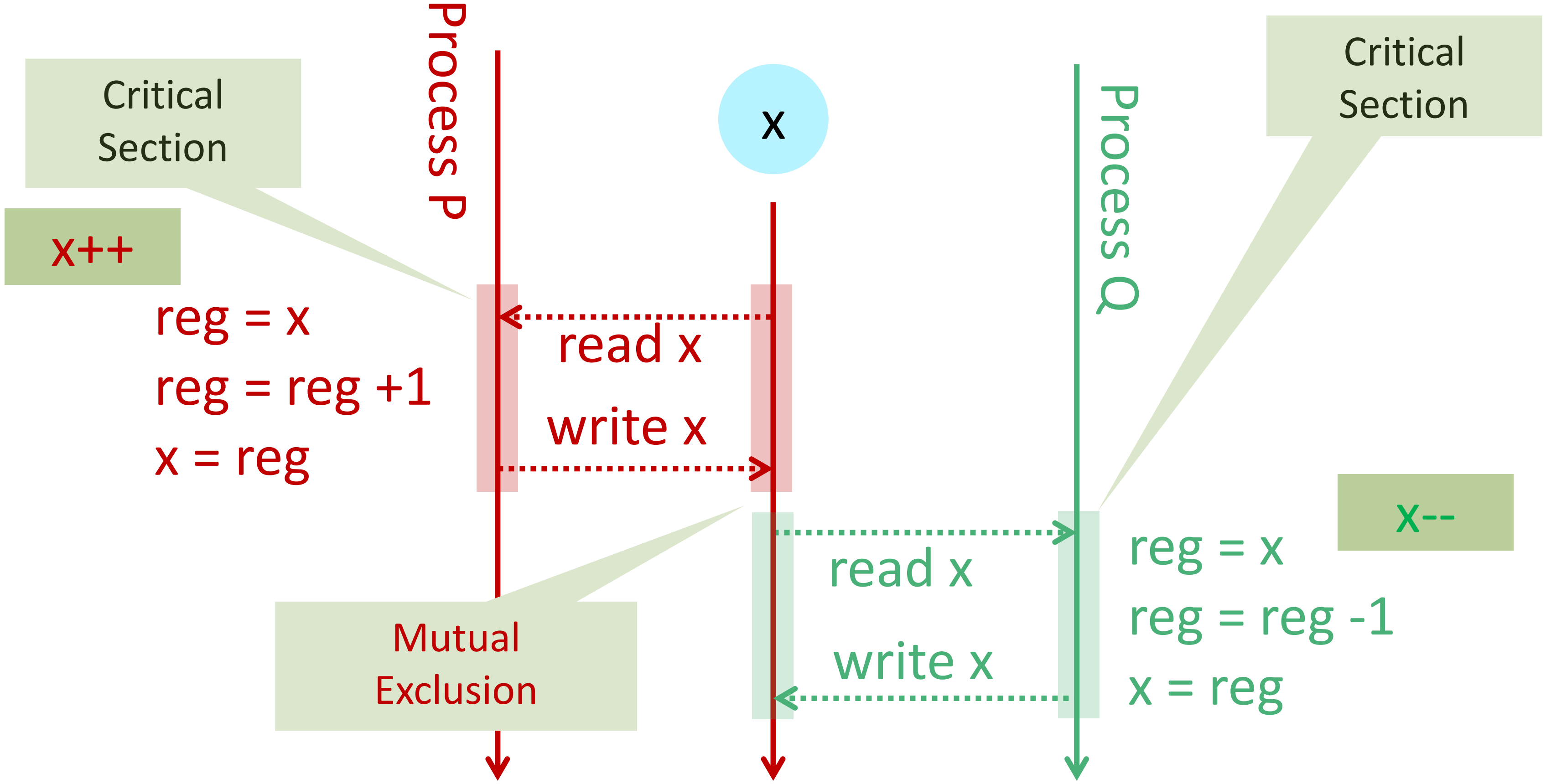
«count provides the number of processes that have passed the barrier" (violated)

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

Recap: Race Condition on «count++»



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	...			

Does this work for one iteration?

What is the value of count and barrier after the call?

`acquire(barrier)` `release(barrier)` } 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	...		

Race Condition !

Race Condition !

Dou you see the problem?

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	...		

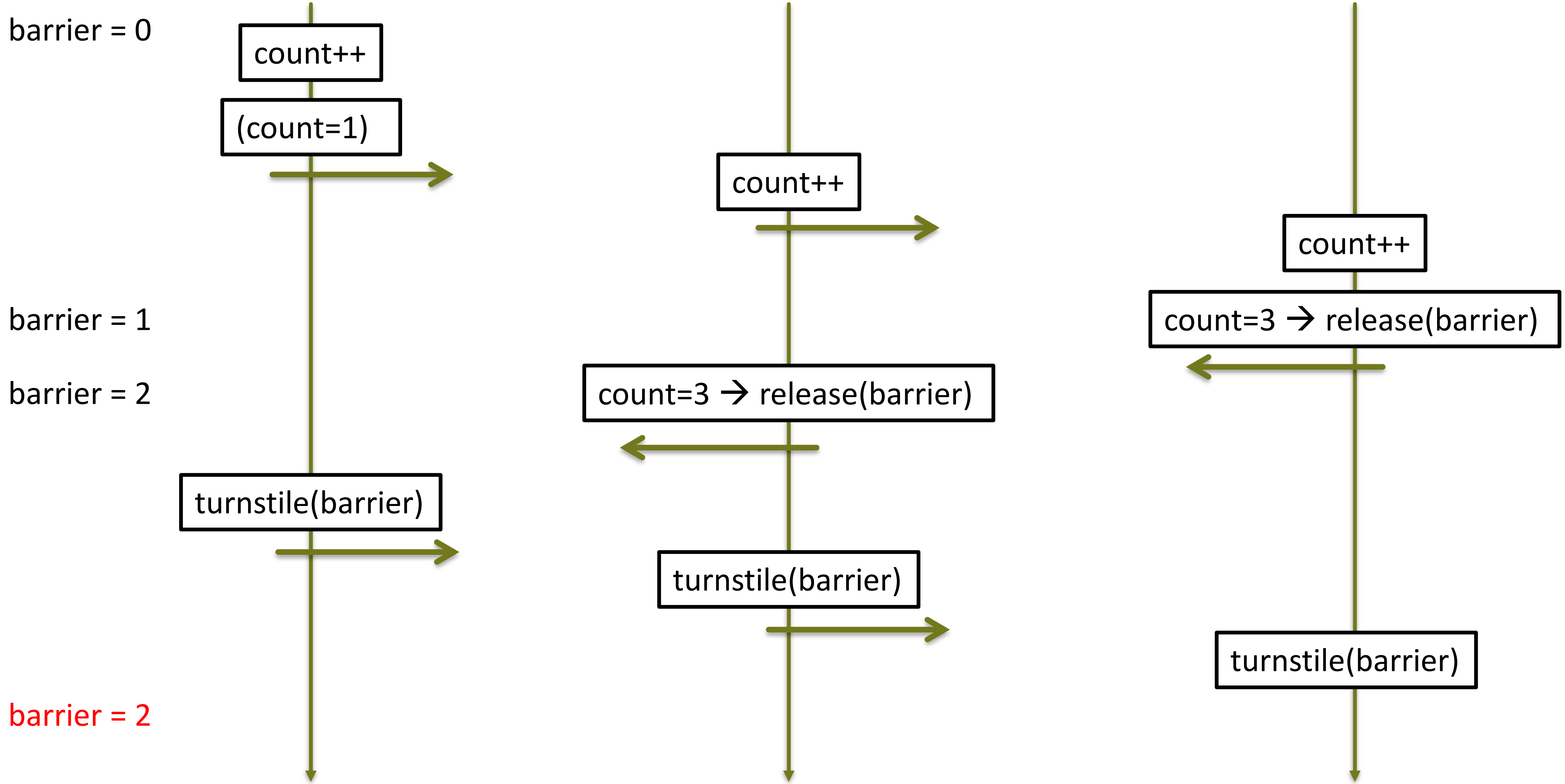
Invariants

«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     | ...
```

n-1 processes here, one process cycles

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)
release(mutex)

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

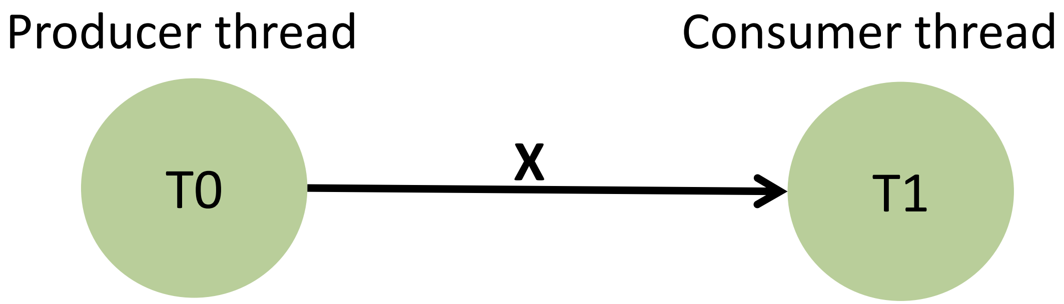
Next:

Producer-Consumer Problem

Monitors and condition variables

Producer Consumer Pattern

Producer / Consumer Pattern



T0 computes X and passes it to T1

T1 uses X

Is synchronization for X needed?

- No because, at any point in time only one thread accesses X
- we, however, need a synchronized mechanism to pass X from T0 to T1

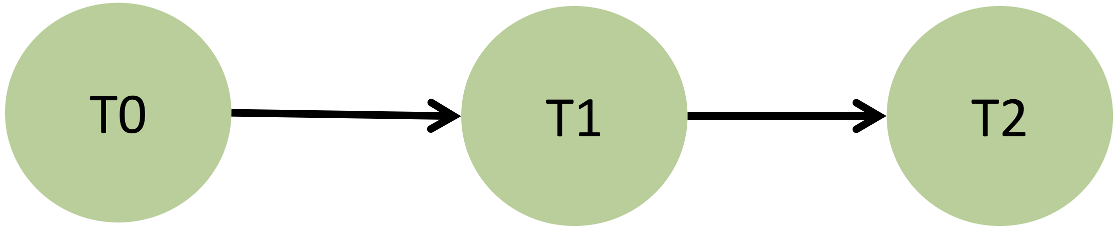
Producer / Consumer Pattern

Fundamental parallel programming pattern

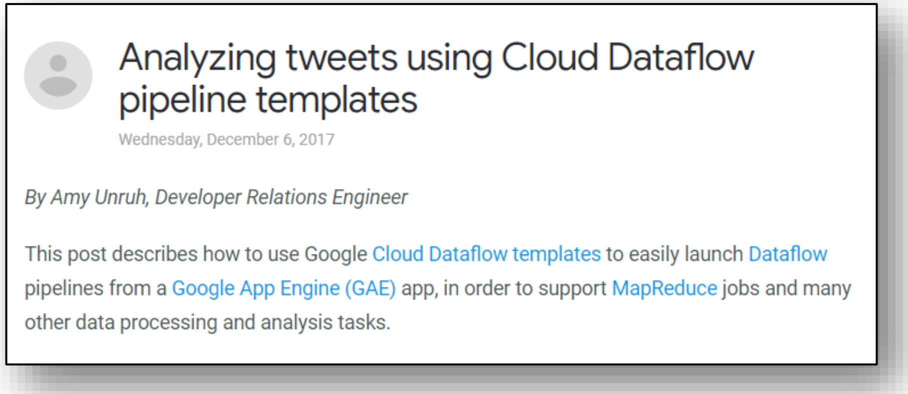
Can be used to build data-flow parallel programs

E.g., pipelines:

30 billion (30 * 10^9) transistors, programmable at fine-grain!



both producer and consumer



Analyzing tweets using Cloud Dataflow pipeline templates

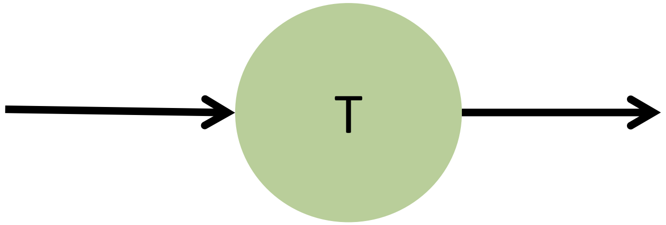
Wednesday, December 6, 2017

By Amy Unruh, Developer Relations Engineer

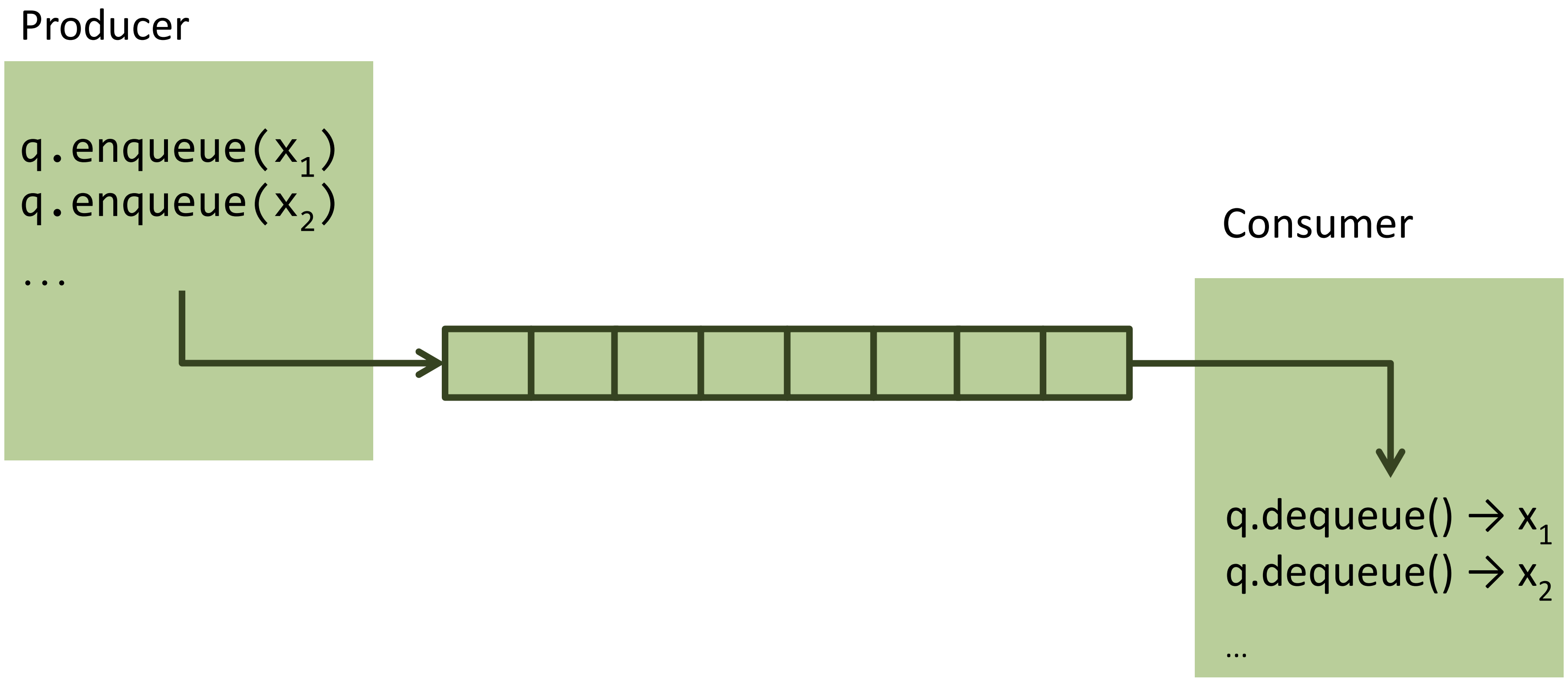
This post describes how to use Google Cloud Dataflow templates to easily launch Dataflow pipelines from a Google App Engine (GAE) app, in order to support MapReduce jobs and many other data processing and analysis tasks.

Pipeline Node

```
while (true) {
   input = q_in.dequeue();
   output = do_something(input);
   q_out.enqueue(output)
}
```



Producer / Consumer queues



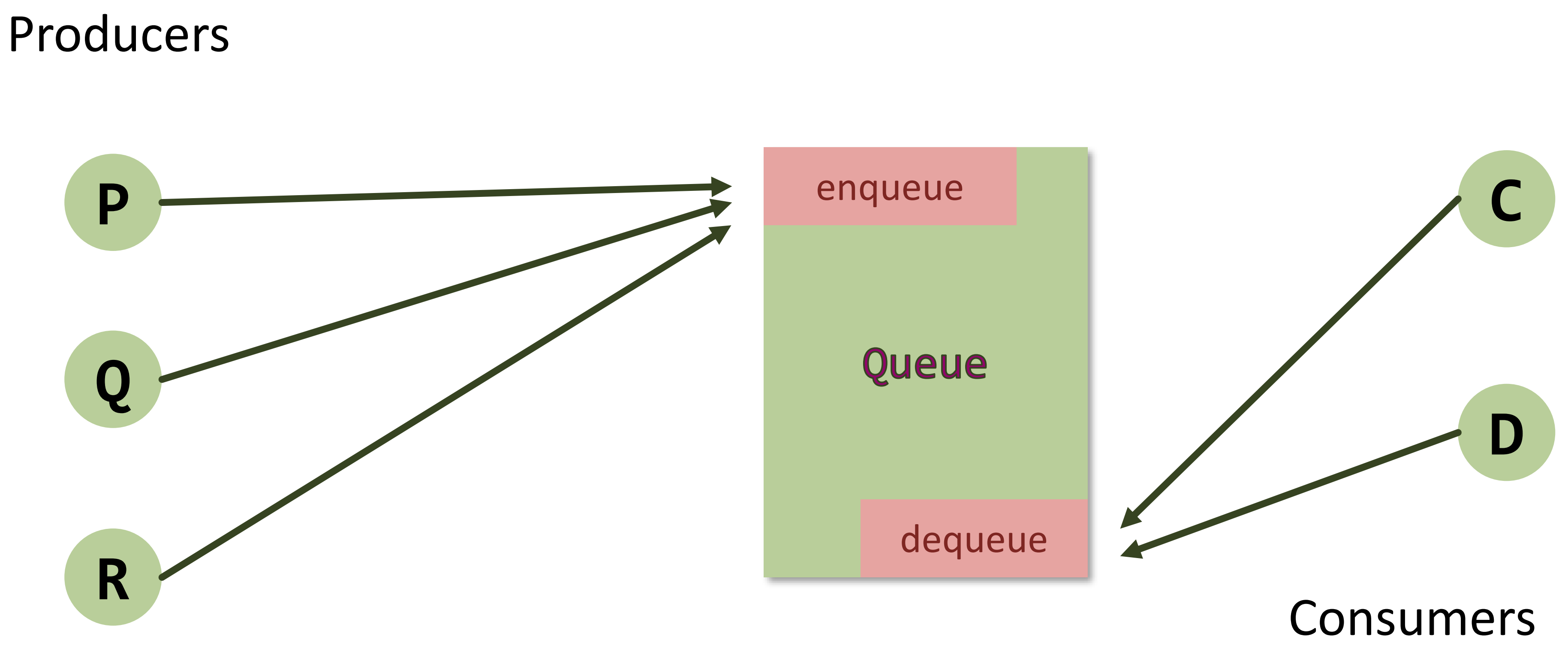
Producer

```
q.enqueue(x₁)
q.enqueue(x₂)
...
```

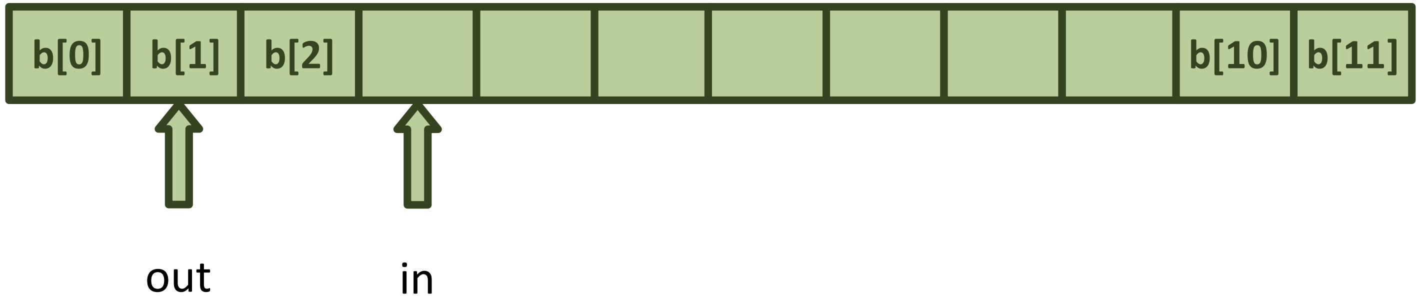
Consumer

q.dequeue() → x_1
q.dequeue() → x_2
…

Multiple Producers and Consumers

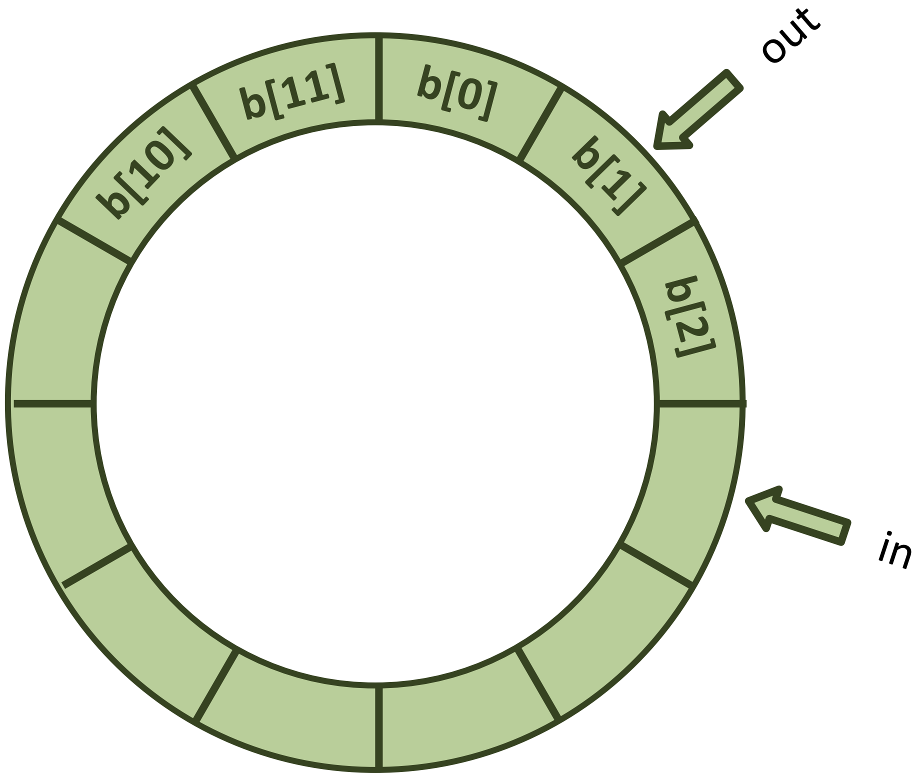


Bounded FIFO as Circular Buffer

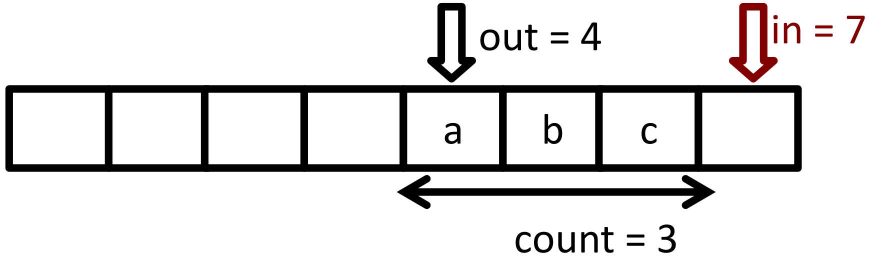


+ wrap around semantics

=



Producer / Consumer queue implementation



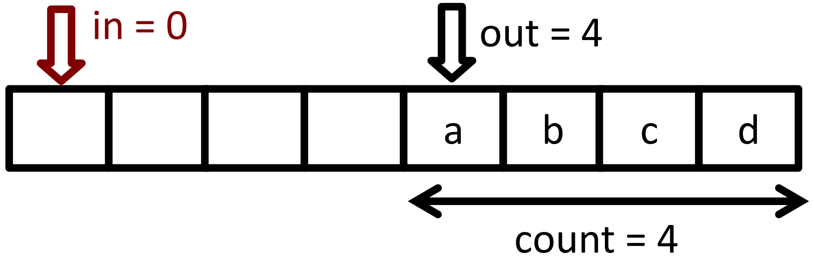
out = 4

in = 7

				a	b	c	

count = 3

Enqueue d



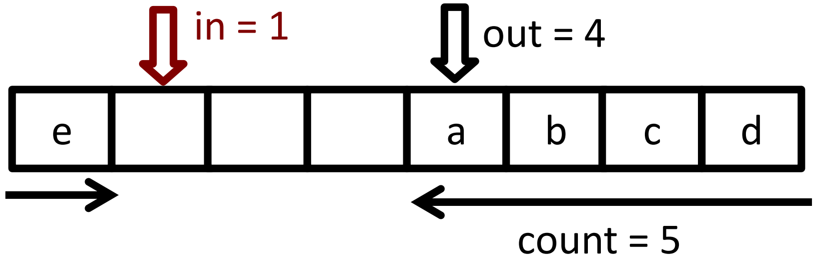
in = 0

out = 4

				a	b	c	d

count = 4

Enqueue e

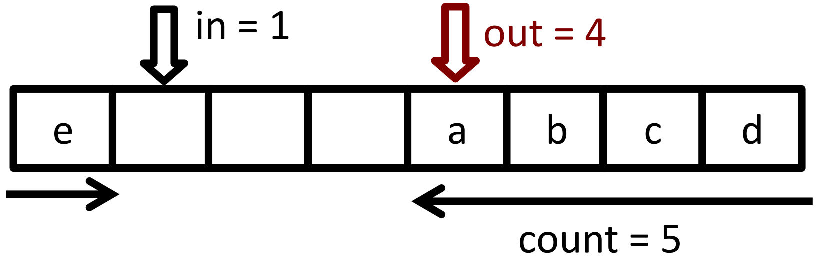


in = 1

out = 4

e				a	b	c	d

count = 5



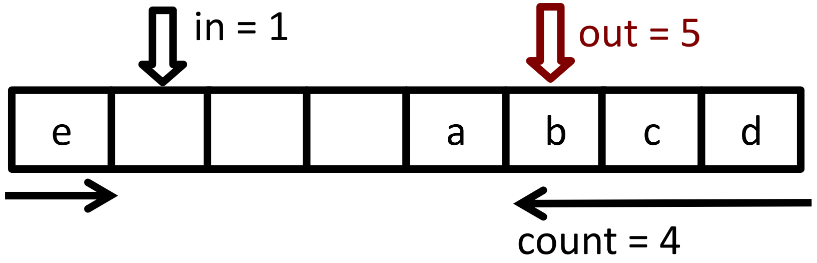
in = 1

out = 4

e				a	b	c	d

count = 5

Dequeue → a



in = 1

out = 5

e				a	b	c	d

count = 4

Producer / Consumer queue implementation

```
class Queue {
  private int in; // next new element
  private int out; // next element
  private int size; // queue capacity
  private long[] buffer;

  Queue(int size) {
    this.size = size;
    in = out = 0;
    buffer = new long[size];
  }

  private int next(int i) {
    return (i + 1) % size;
  }
```

```
public synchronized void enqueue(long item) {
   buffer[in] = item;
   in = next(in);
}
public synchronized long dequeue() {
   item = buffer[out];
   out = next(out);
   return item;
}
```

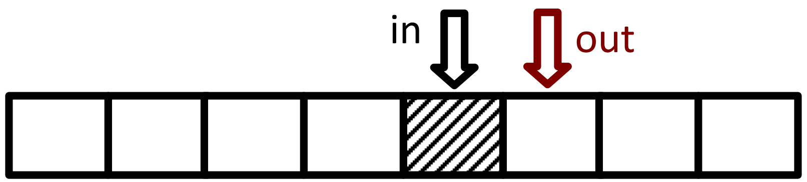
What if we try to

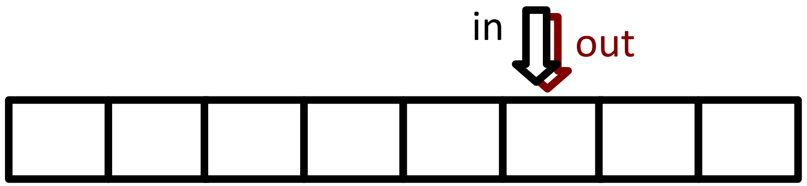
1. dequeue from an empty queue?
2. enqueue to a full queue?

Producer / Consumer queues: helper functions

```
public void doEnqueue(long item) {
  buffer[in] = item;
  in = next(in);
}

public boolean isFull() {
  return (in+1) % size == out;
}
```

```
public long doDequeue() {
  long item = buffer[out];
  out = next(out);
  return item;
}
public boolean isEmpty() {
  return in == out;
}
```





full: one element not usable.
Still it has a benefit to not use a counter variable. Any idea what this benefit could be?

Producer / Consumer queues

```
public synchronized void enqueue(long item) {
      while (isFull())
         ; // wait
      doEnqueue(item);
}
```

```
public void doEnqueue(long item) {
  buffer[in] = item;
  in = next(in);
}
public boolean isFull() {
  return (in+1) % size == out;
}
```

```
public synchronized long dequeue() {
      while (isEmpty())
         ; // wait
      return doDequeue();
}
```

```
public long doDequeue() {
  long item = buffer[out];
  out = next(out);
  return item;
}
public boolean isEmpty() {
  return in == out;
}
```

Do you see the problem?

→ **Blocks forever**

infinite loops with a lock held …

Producer / Consumer queues using sleep()

```
public void enqueue(long item) throws InterruptedException {
  while (true) {
    synchronized(this) {
      if (!isFull()) {
        doEnqueue(item);
        return;
      }
    }
    Thread.sleep(timeout); // sleep without lock!
  }
}
```

What is the proper value for the timeout?
Ideally we would like to be notified when the change happens!
When is that?

Producer / Consumer queues with semaphores

```
import java.util.concurrent.Semaphore;

class Queue {
    int in, out, size;
    long buf[];
    Semaphore nonEmpty, nonFull, manipulation;

    Queue(int s) {
        size = s;
        buf = new long[size];
        in = out = 0;
        nonEmpty = new Semaphore(0); // use the counting feature of semaphores!
        nonFull = new Semaphore(size); // use the counting feature of semaphores!
        manipulation = new Semaphore(1); // binary semaphore
    }
}
```

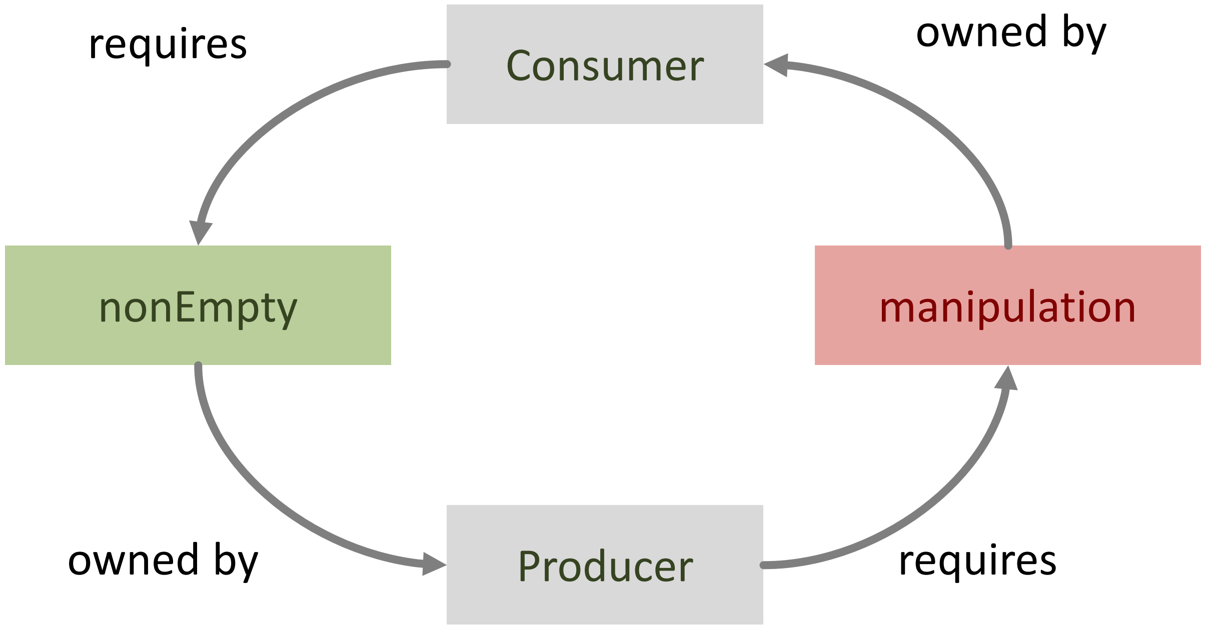
Producer / Consumer queues with semaphores, correct?

Do you see the problem?

```
void enqueue(long x) {

    try {
        manipulation.acquire();
        nonFull.acquire();
        buf[in] = x;
        in = (in+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        manipulation.acquire();
        nonEmpty.acquire();
        x = buf[out];
        out = (out+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

Deadlock!


requires
Consumer
owned by
nonEmpty
manipulation
owned by
Producer
requires

Producer / Consumer queues with semaphores

```
void enqueue(long x) {

    try {
        nonFull.acquire();
        manipulation.acquire();
        buf[in] = x;
        in = next(in);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        nonEmpty.acquire();
        manipulation.acquire();
        x = buf[out];
        out = next(out);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

Why are semaphores (and locks) problematic?

Semaphores are unstructured. Correct use requires high level of discipline.

Easy to introduce deadlocks with semaphores.

We need: a lock that we can temporarily escape from when waiting on a condition.

Monitors

Monitors

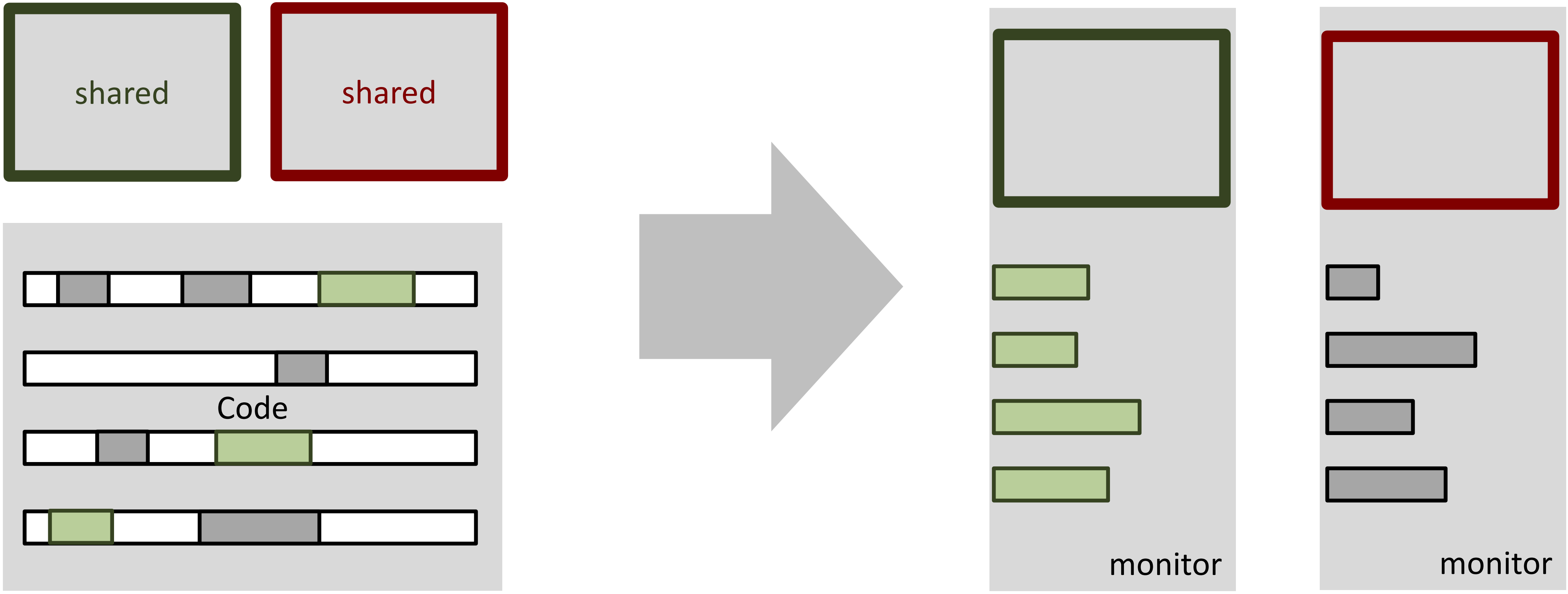
Monitor:
abstract data structure equipped with a set of operations that run in mutual exclusion.

Invented by Tony Hoare and Per Brinch Hansen (cf. Monitors: An Operating System Structuring Concept, Tony Hoare, 1974)

Tony Hoare
(1934-today)

Per Brinch Hansen
(1938-2007)

Monitors vs. Semaphores/Unbound Locks



Producer / Consumer queues

```
public void synchronized enqueue(long item) {
   "while (isFull()) wait"
   doEnqueue(item);
}
```

```
public long synchronized dequeue() {
   "while (isEmpty()) wait"
   return doDequeue();
}
```

The mutual exclusion part is nicely available already.
But: while the buffer is full we need to give up the lock, how?

Monitors

Monitors provide, in addition to mutual exclusion, a mechanism to check conditions with the following semantics:

If a condition does not hold

- Release the monitor lock
- Wait for the condition to become true
- Signaling mechanism to avoid busy-loops (spinning)

Monitors in Java

Uses the intrinsic lock (`synchronized`) of an object

+ `wait / notify / notifyAll`:

`wait()` – the current thread waits until it is signaled (via notify)

`notify()` – wakes up *one* waiting thread (an arbitrary one)

`notifyAll()` – wakes up *all* waiting threads

Producer / Consumer with monitor in Java

```
class Queue {
     int in, out, size;
     long buf[];

     Queue(int s) {
          size = s;
          buf = new long[size];
          in = out = 0;
     }
     ...
}
```

Producer / Consumer with monitor in Java

```
synchronized void enqueue(long x) {

    while (isFull())
        try {
            wait();
        } catch (InterruptedException e) { }
    doEnqueue(x);
    notifyAll();
}
```

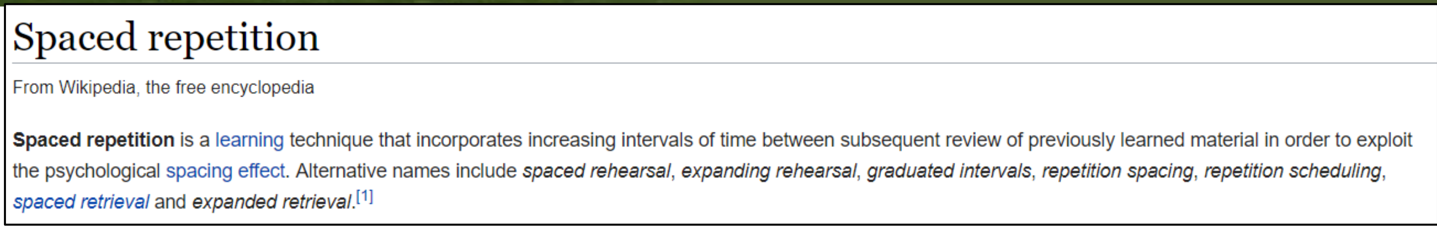
```
synchronized long dequeue() {
    long x;
    while (isEmpty())
        try {
            wait();
        } catch (InterruptedException e) { }
    x = doDequeue();
    notifyAll();
    return x;
}
```

Wouldn't an if be sufficient?

(Why) can't we use notify()?

IMPORTANT TO KNOW JAVA MONITOR IMPLEMENTATION DETAILS

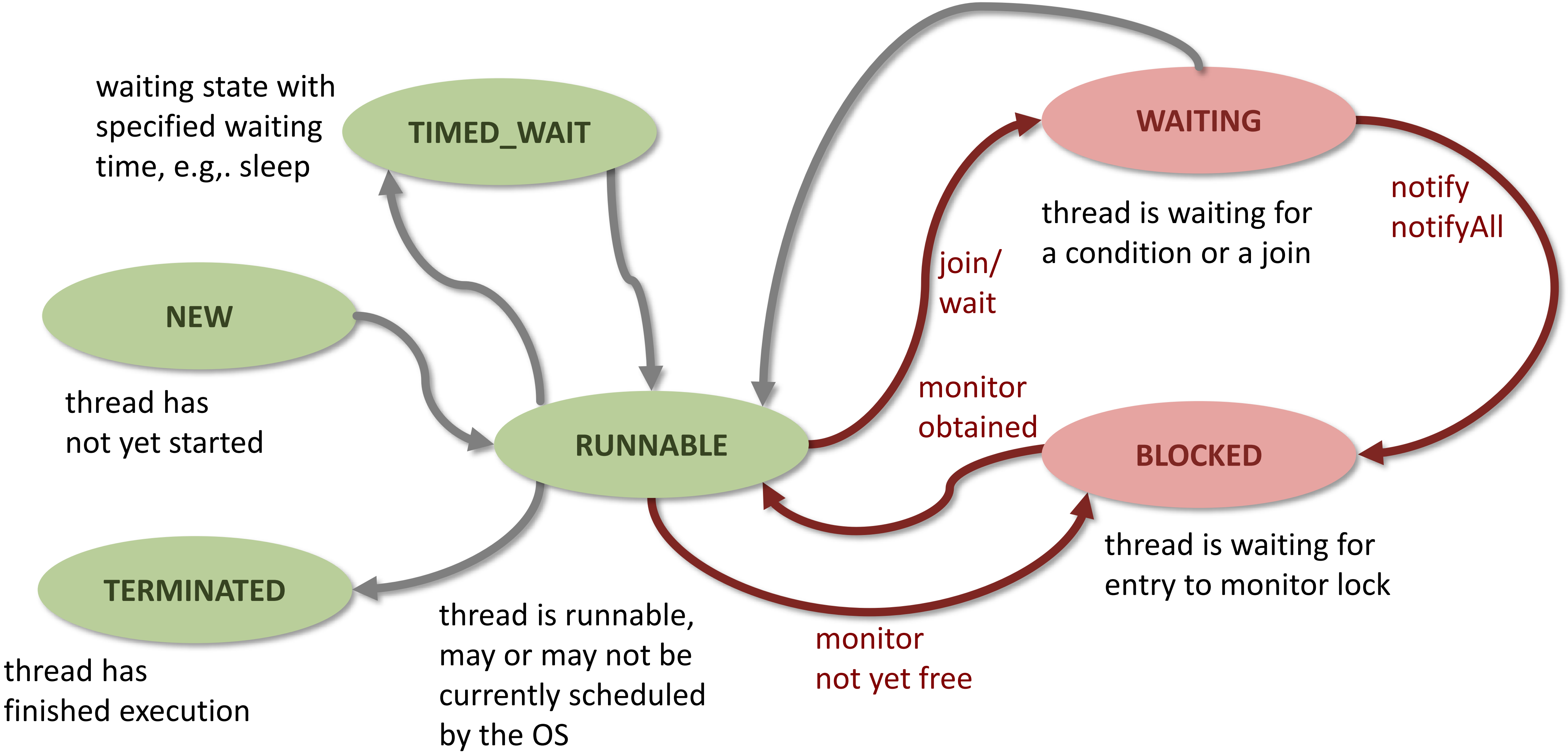
Thread States in Java



Spaced repetition

From Wikipedia, the free encyclopedia

Spaced repetition is a learning technique that incorporates increasing intervals of time between subsequent review of previously learned material in order to exploit the psychological spacing effect. Alternative names include *spaced rehearsal*, *expanding rehearsal*, *graduated intervals*, *repetition spacing*, *repetition scheduling*, *spaced retrieval* and *expanded retrieval*.[1]



waiting state with specified waiting time, e.g,. sleep

TIMED_WAIT

NEW

thread has not yet started

RUNNABLE

TERMINATED

thread has finished execution

thread is runnable, may or may not be currently scheduled by the OS

WAITING

thread is waiting for a condition or a join

join/ wait

notify notifyAll

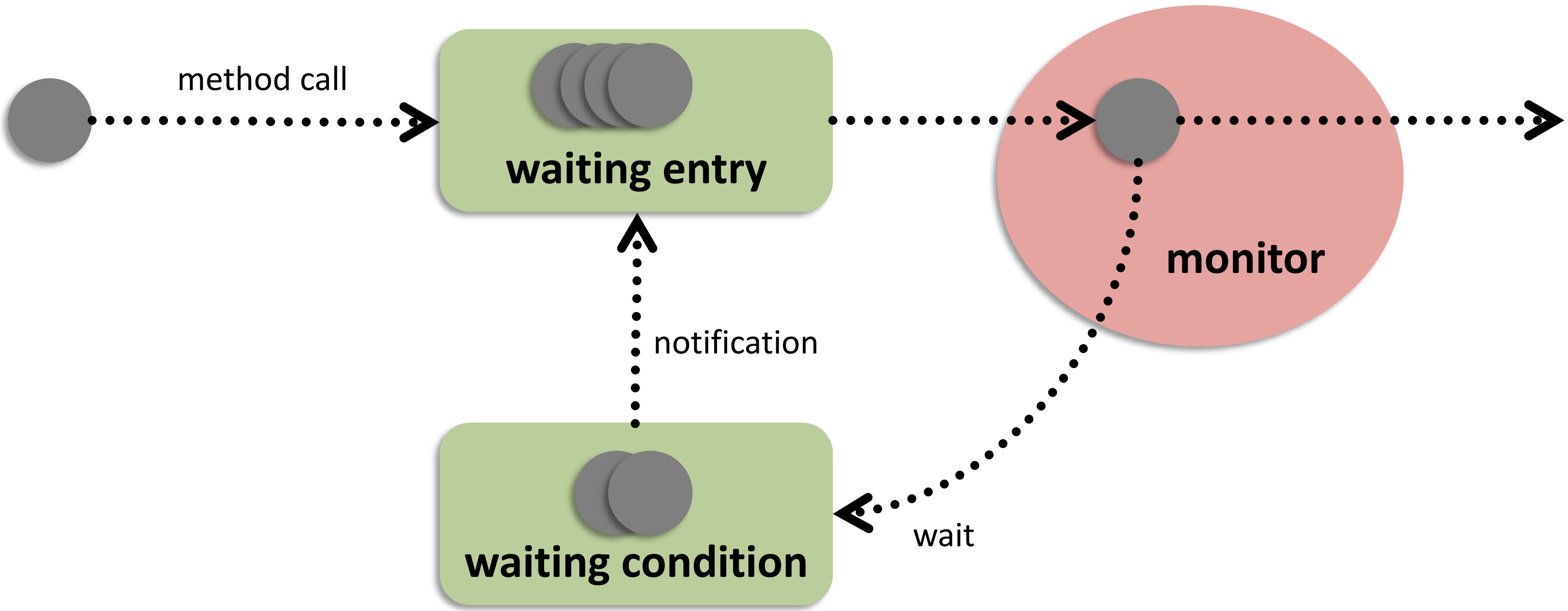
monitor obtained

BLOCKED

thread is waiting for entry to monitor lock

monitor not yet free

Monitor Queues



Exact Semantics

Important to know for the programmer (you): what happens upon notification? Priorities?

signal and wait

- signaling process exits the monitor (goes to waiting entry queue)
- signaling process passes monitor lock to signaled process

signal and continue

- signaling process continues running
- signaling process moves signaled process to waiting entry queue

other semantics: signal and exit, signal and urgent wait …

Why this is important? Let's try this implementing a semaphore:

```
class Semaphore {
   int number = 1; // number of threads allowed in critical section

   synchronized void acquire() {
      if (number <= 0)
         try { wait();   } catch (InterruptedException e) { };
      number--;
   }

   synchronized void release() {
      number++;
      if (number > 0)
         notify();
   }
}
```

Looks good, doesn't it?
But there is a problem.
Do you know which?

Java Monitors = signal + continue

```
R synchronized void acquire() {
    if (number <= 0)
       try { wait();  } Q
       catch (InterruptedException e) { };
    number--;
 }

 synchronized void release() {
P   number++;
    if (number > 0)
       notify();
 }
```

Scenario:

1. **Process P has previously entered the semaphore and decreased number to 0.**
2. **Process Q sees number = 0 and goes to waiting list.**
3. **P is executing exit. In this moment process R wants to enter the monitor via method enter.**
4. **P signals Q and thus moves it into wait entry list (signal and continue!). P exits the function/lock.**
5. **R gets entry to monitor before Q and sees the number = 1**
6. **Q continues execution with number = 0!**

Inconsistency!

The cure – a while loop.

```
synchronized void acquire() {
 while (number <= 0)
  try { wait(); }
   catch (InterruptedException e) { };
 number--;
}
```

```
synchronized void release() {
   number++;
   if (number > 0)
     notify();
}
```

If, additionally, different threads evaluate different conditions, the notification has to be a `notifyAll`. In this example it is not required.

Something different: Java Interface Lock

Intrinsic locks ("synchronized") with objects provide a good abstraction and should be first choice

Limitations

- **one implicit lock per object**
- **are forced to be used in blocks**
- **limited flexibility**

Java offers the Lock interface for more flexibility (e.g., lock can be polled).

```
final Lock lock = new ReentrantLock();
```

Condition interface

Java Locks provide *conditions that can be instantiated*

```
Condition notFull  = lock.newCondition();
```

Java conditions offer

.await() – the current thread waits until condition is signaled

.signal() – wakes up one thread *waiting on this condition*

.signalAll() – wakes up all threads *waiting on this condition*

Condition interface

→ **Conditions are always associated with a lock**
lock.newCondition()

.await()

- called with the lock held
- **atomically** releases the lock and waits until thread is signaled
- when returns, it is **guaranteed** to hold the lock
- thread **always** needs to check condition

.signal{,All}() – wakes up one (all) waiting thread(s)

- called with the lock held

Producer / Consumer with explicit Lock

```
class Queue{
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    final Condition notFull  = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s;
        buf = new long[size];
    }

...

}
```

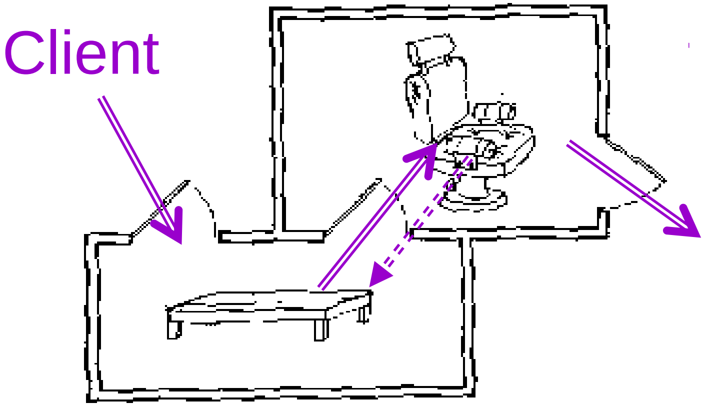
Producer / Consumer with Lock

```
void enqueue(long x){

   lock.lock();
   while (isFull())
      try {
         notFull.await();
      } catch (InterruptedException e){}
   doEnqueue(x);
   notEmpty.signal();
   lock.unlock();
}
```

```
long dequeue() {
   long x;
   lock.lock();
   while (isEmpty())
      try {
         notEmpty.await();
      } catch (InterruptedException e){}
   x = doDequeue();
   notFull.signal();
   lock.unlock();
   return x;
}
```

The Sleeping Barber Variant (E. Dijkstra)

Disadvantage of the solution: nonfull and nonempty signal will be sent in any case, even when no threads are waiting.



Sleeping barber **variant: additional counters for checking if processes are waiting:**

$m \leq 0 \Leftrightarrow$ **buffer full & -m producers (clients) are waiting**

$n \leq 0 \Leftrightarrow$ **buffer empty & -n consumers (barbers) are waiting**

Producer Consumer, Sleeping Barber Variant

```
class Queue{
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    int n = 0; final Condition notFull  = lock.newCondition();
    int m; final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s; m=size-1;
        buf = new long[size];
    }
    ...
}
```

sic! cf. unused element in original producer/consumer queue!

Producer Consumer, Sleeping Barber Variant

```
void enqueue(long x) {

   lock.lock();
   m--; if (m<0)
      while (isFull())
         try { notFull.await(); }
         catch(InterruptedException e){}
   doEnqueue(x);
   n++;
   if (n<=0) notEmpty.signal();
   lock.unlock();

}
```

```
long dequeue() {
   long x;
   lock.lock();
   n--; if (n<0)
      while (isEmpty())
         try { notEmpty.await(); }
         catch(InterruptedException e){}
   x = doDequeue();
   m++;
   if (m<=0) notFull.signal();
   lock.unlock();
   return x;
}
```

Guidelines for using condition waits

- **Always have a condition predicate**
- **Always test the condition predicate:**
  - before calling wait
  - after returning from wait
- **Always call wait in a loop**
- **Ensure state is protected by lock associated with condition**

java.concurrent.util

Java (luckily for us) provides many common synchronization objects:

- **Semaphores**
- **Barriers (CyclicBarrier)**
- **Producer / Consumer queues**
- **and many more... (Latches, Futures, ...)**