

TORSTEN HOEFLER

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

EU ministers commit to digitising Europe with high-performance computing power

Published on 23/03/2017

Ministers from seven European countries (France, Germany, Italy, Luxembourg, Netherlands, Portugal and Spain) have signed in Rome a declaration to support the next generation of computing and data infrastructures, a European project of the size of Airbus in the 1990s and of Galileo in the 2000s.



© Thinkstock

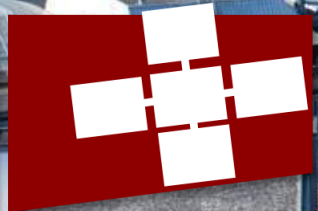


European Processor Initiative

They plan to establish EuroHPC for acquiring and deploying an integrated world-class high-performance computing infrastructure capable of at least 10^{18} calculations per second (so-called exascale computers). This will be available across the EU for scientific communities, industry and the public sector, no matter where the users are located.

Andrus Ansip, European Commission Vice-President for the Digital Single Market welcomed this important step: "High-performance computing is moving towards its next frontier - more than 100 times faster than the fastest machines currently available in Europe. But not all EU countries have the capacity to build and maintain such infrastructure, or to develop such technologies on their own. If we stay dependent on others for this critical resource, then we risk getting technologically 'locked', delayed or deprived of strategic know-how. Europe needs integrated world-class capability in supercomputing to be ahead in the global race. Today's declaration is a great step forward. I encourage even more EU countries to engage in this ambitious endeavour". See full speech by Vice-President Ansip at the [Digital Day](#) in Rome.

High-performance computing (HPC) involves thousands of processors working in parallel to analyse billions of pieces of data in real time. HPC allows to design and new drugs and simulate their effects, and provide faster diagnosis, better treatments and personalised health care. It can make our communications and



SPCL

Another (historic) example: from the Java standard library

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

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

Do you find the two problems?



Another (historic) example: from the Java standard library

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

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

Do you find the two problems?

Problem #1:

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

Problem #2:

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

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 unsuccessful 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. σμα = 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 $s \geq 0$ and the following operations*

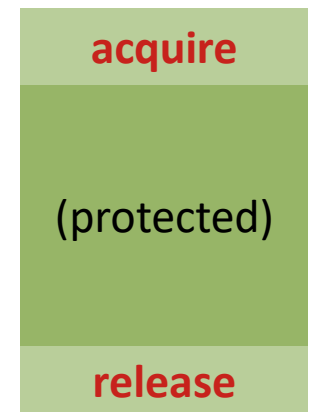
acquire(S)

atomic {
 wait until $S > 0$
 dec(S)
 }



release(S)

atomic {
 inc(S)
 }



* 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

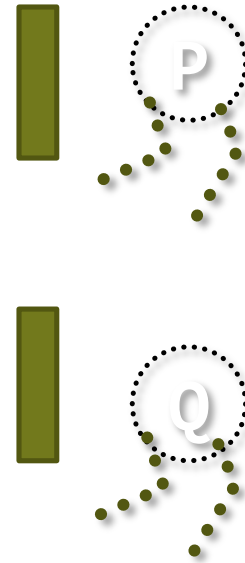
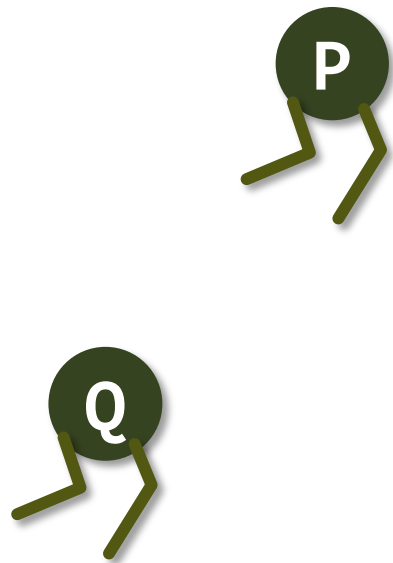
$x > 0$ → x threads will be let into “critical section”

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?

Rendezvous with Semaphores

- Two processes P and Q executing code.
- Rendezvous: 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
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	?	?
<i>post</i>

Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)

Semaphores **P_Arrived** and **Q_Arrived**

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) ?	acquire(P_Arrived) ?
<i>post</i>

Rendezvous with Semaphores

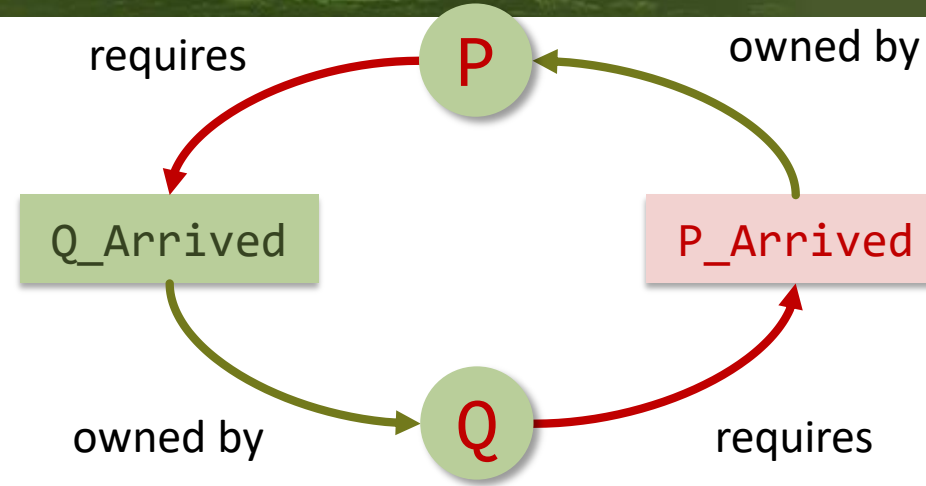
Synchronize Processes P and Q at one location (Rendezvous)

Semaphores **P_Arrived** and **Q_Arrived**

Dou you find the problem?

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	acquire(Q_Arrived) release(P_Arrived)	acquire(P_Arrived) release(Q_Arrived)
<i>post</i>

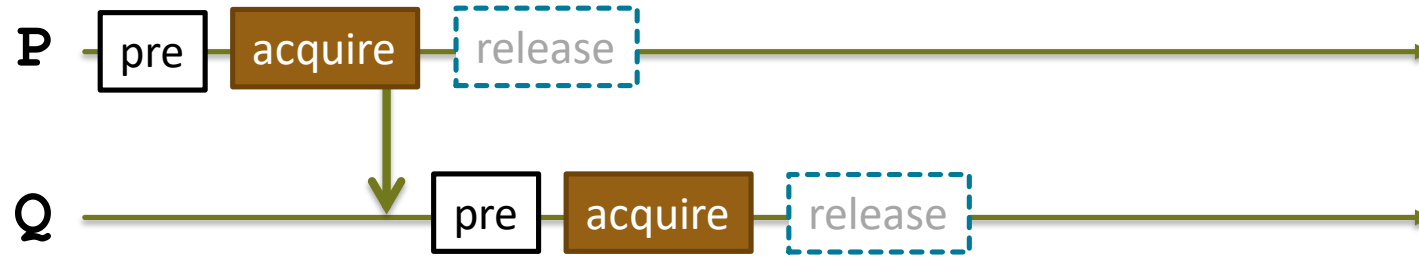
Deadlock



	P	Q
<i>init</i>	<code>P_Arrived=0</code>	<code>Q_Arrived=0</code>
<i>pre</i>
<i>rendezvous</i>	<code>acquire(Q_Arrived)</code> <code>release(P_Arrived)</code>	<code>acquire(P_Arrived)</code> <code>release(Q_Arrived)</code>
<i>post</i>

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
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) acquire(Q_Arrived)	acquire(P_Arrived) release(Q_Arrived)
<i>post</i>

Implementing Semaphores without Spinning (blocking queues)

Consider a process list Q_s associated with semaphore S

acquire(S)

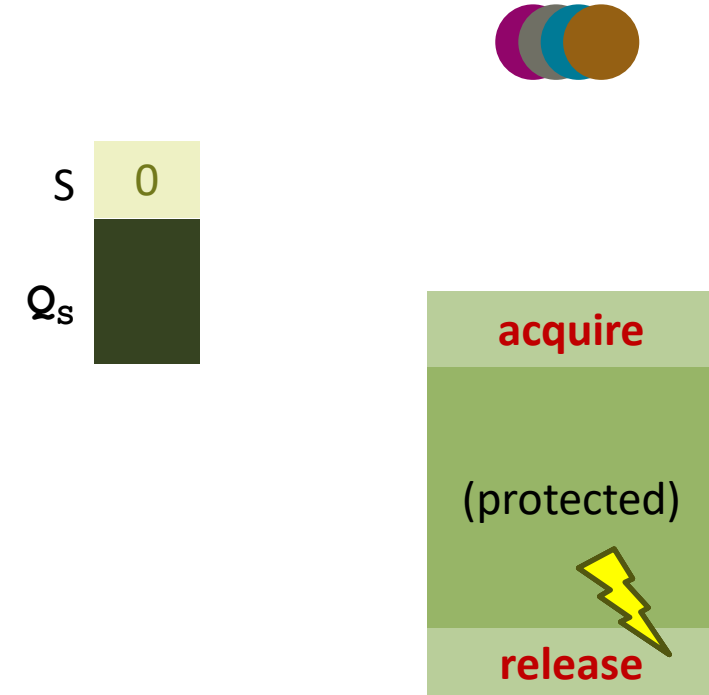
atomic

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

release(S)

atomic

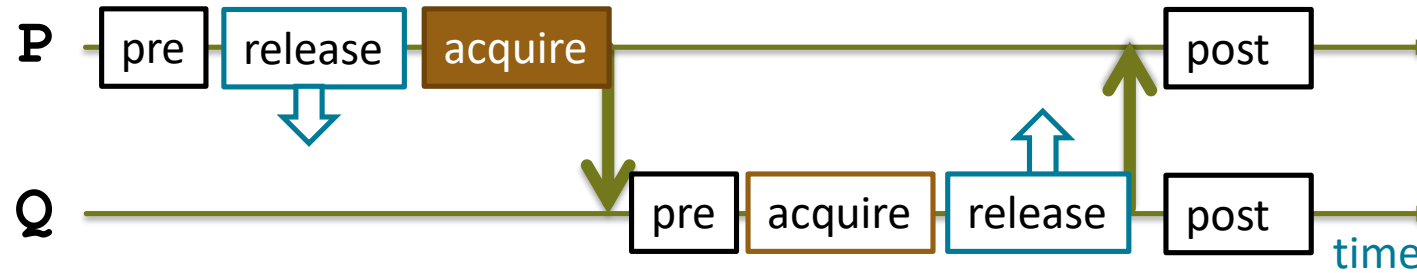
```
{if  $Q_s == \emptyset$  then
    inc( $S$ )
else
    get( $Q_s$ , p)
    unblock(p)
end }
```



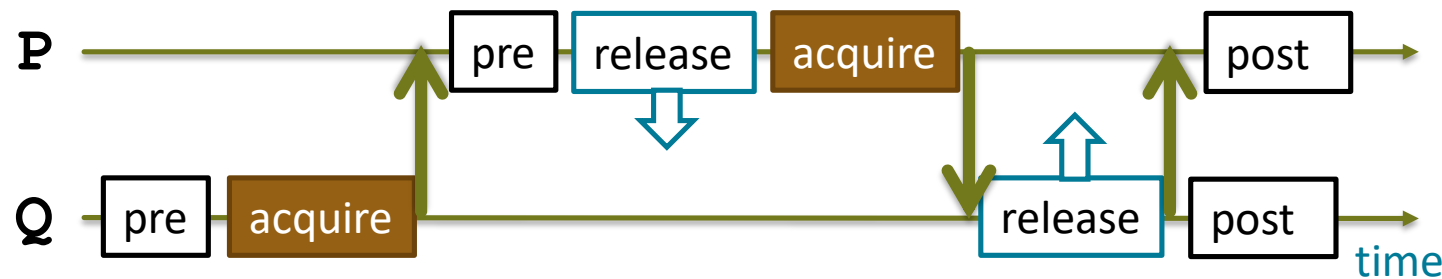
Scheduling Scenarios

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) acquire(Q_Arrived)	acquire(P_Arrived) release(Q_Arrived)
<i>post</i>

P first



Q first



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

Rendezvous with Semaphores

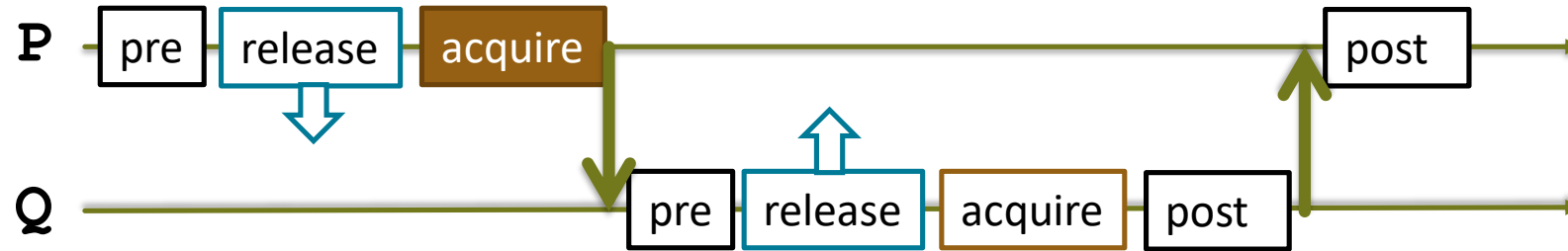
Synchronize Processes P and Q at one location (Rendezvous)

Assume Semaphores **P_Arrived** and **Q_Arrived**

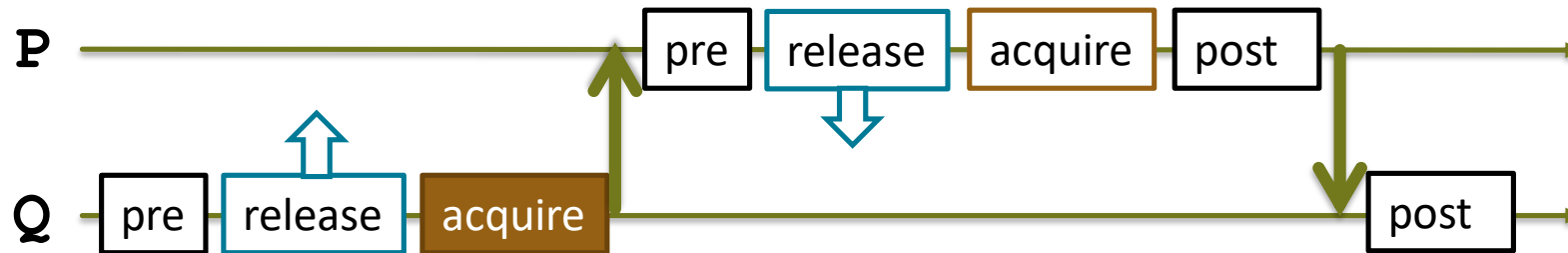
	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) acquire(Q_Arrived)	release(Q_Arrived) acquire(P_Arrived)
<i>post</i>

That's even better.

P first



Q first



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

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) acquire(Q_Arrived)	release(Q_Arrived) acquire(P_Arrived)
<i>post</i>

Back to our dot-product

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

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



A. Bridging Model for parallel Computation

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

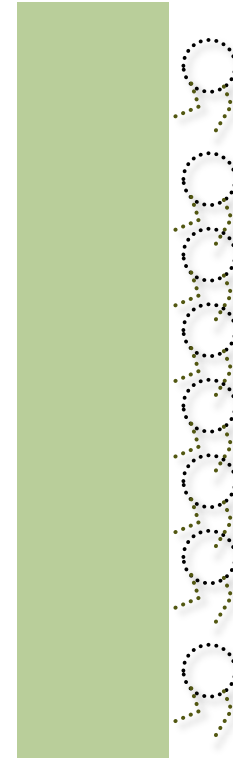
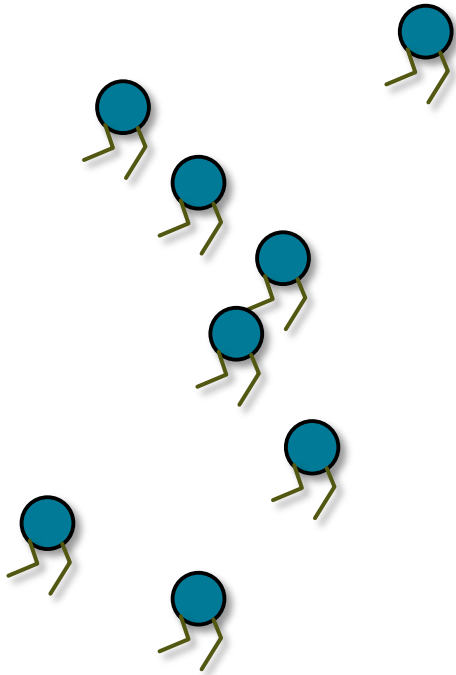
Leslie G. Valiant



Barriers

Barrier

Synchronize a number of processes.

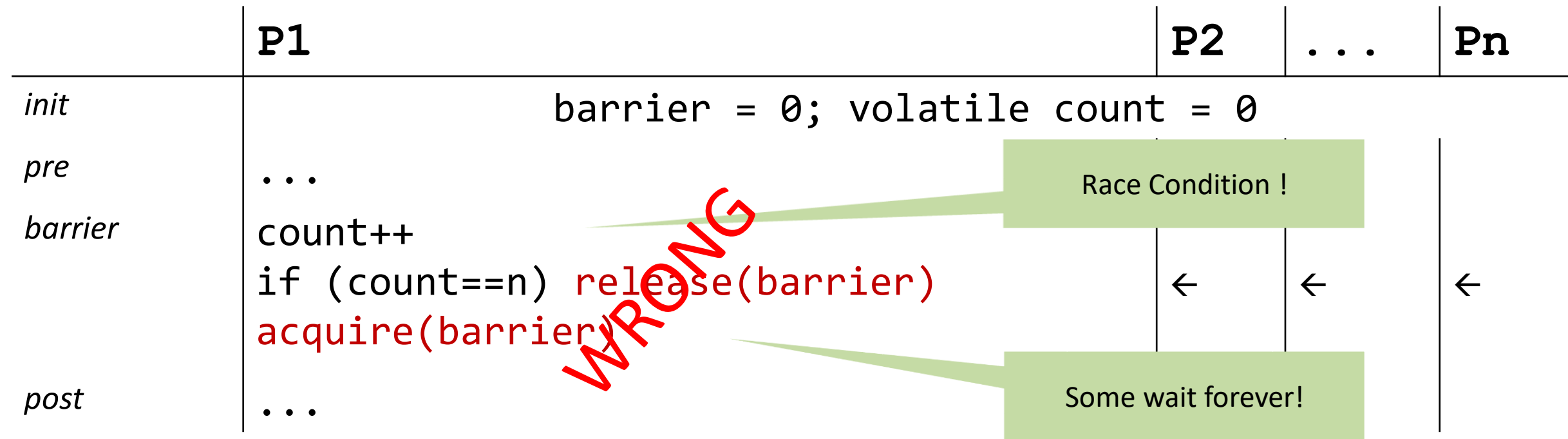


How would you implement this using Semaphores?

Barrier – 1st try

Synchronize a number (n) of processes.

Semaphore **barrier**. Integer count.



Barrier

Synchronize a number (n) of processes.

Semaphore **barrier**. Integer count.

	P1
<i>init</i>	<code>barrier = 0; volatile count</code>
<i>pre</i>	<code>...</code>
<i>barrier</i>	<code>count++</code> <code>if (count==n) release(barrier)</code> <code>acquire(barrier)</code>
<i>post</i>	<code>...</code>

WRONG

Invariants

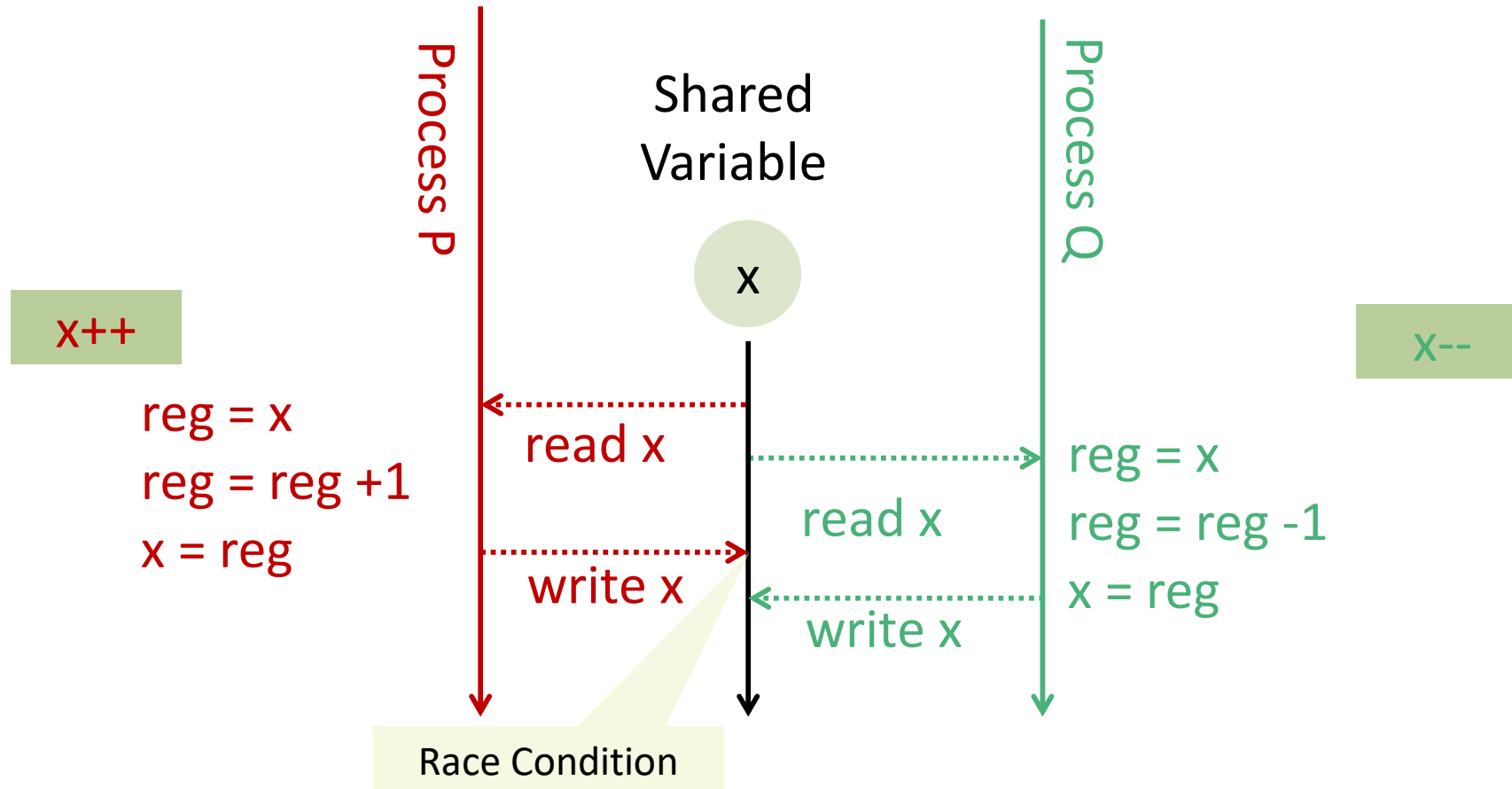
«Each of the processes eventually reaches the acquire statement»

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

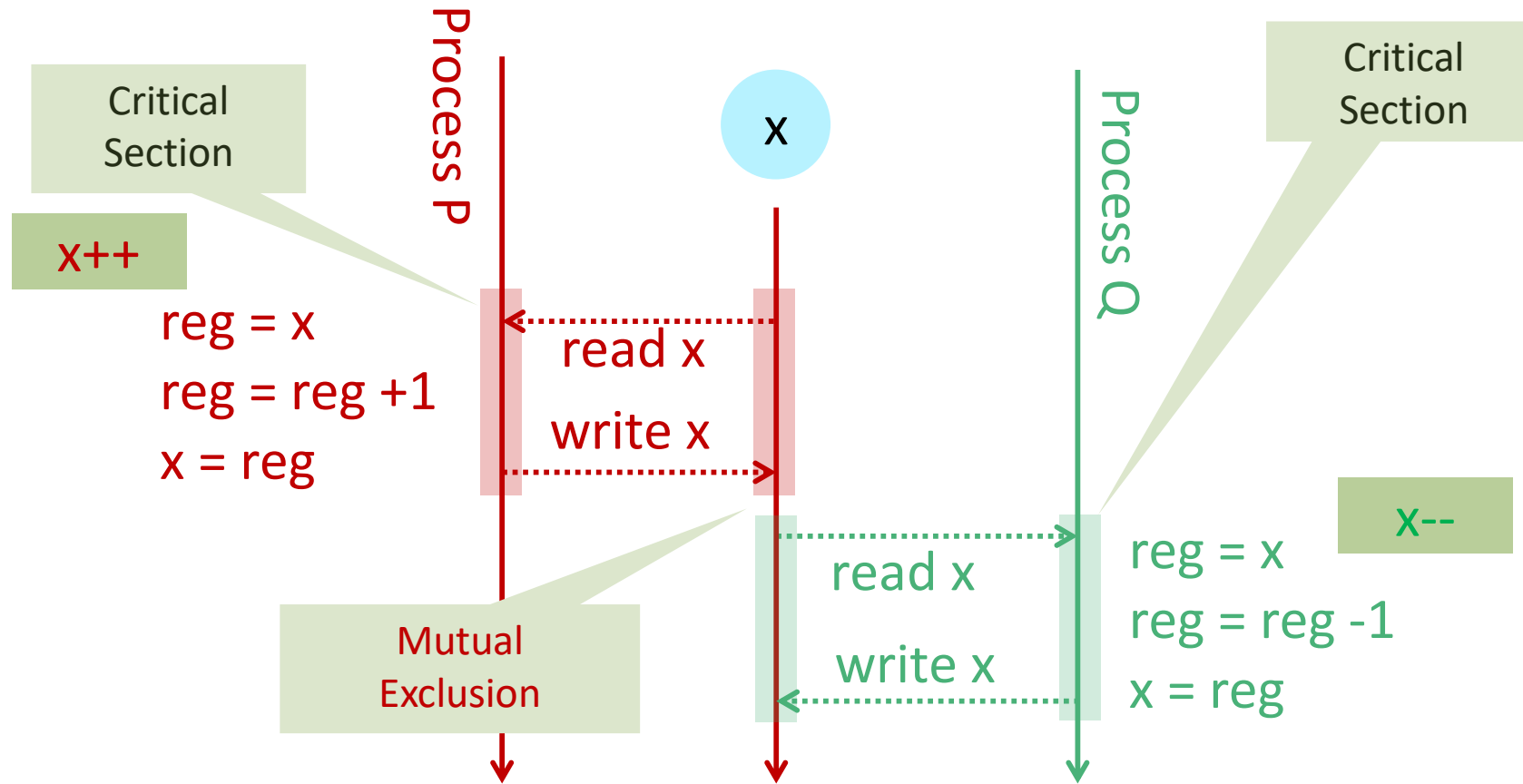
«count provides the number of processes that have passed the barrier» (violated)

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

Recap: Race Condition



With Mutual Exclusion

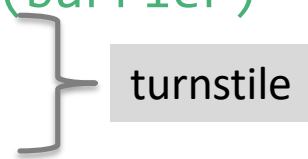


Barrier

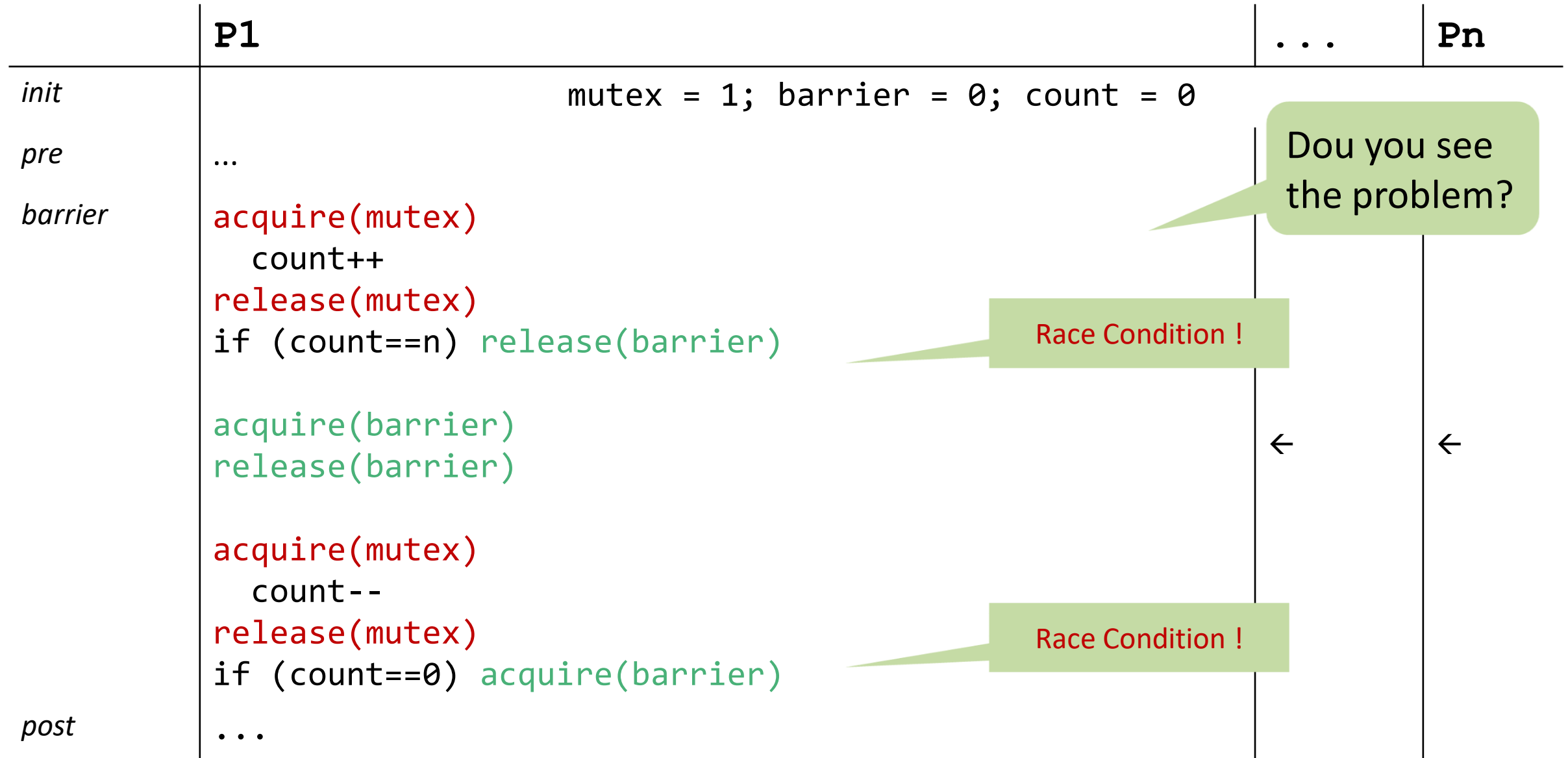
Synchronize a number (n) of processes.

Semaphores **barrier**, **mutex**. Integer count.

	P1	P2	...	Pn
<i>init</i>	mutex = 1; barrier = 0; count = 0			
<i>pre</i>	...			
<i>barrier</i>	acquire(mutex) count++ release(mutex) if (count==n) release(barrier) acquire(barrier) release(barrier)	←	←	←
<i>post</i>	...			



Reusable Barrier. 1st trial.



Reusable Barrier. 1st trial.

	P1	...	Pn
<i>init</i>	<code>mutex = 1; barrier = 0; count = 0</code>		
<i>pre</i>	...		
<i>barrier</i>	<code>acquire(mutex)</code> <code>count++</code> <code>release(mutex)</code> <code>if (count==n) release(barrier)</code> <code>acquire(barrier)</code> <code>release(barrier)</code> <code>acquire(mutex)</code> <code>count--</code> <code>release(mutex)</code> <code>if (count==0) acquire(barrier)</code>		
<i>post</i>	...		

Invariants

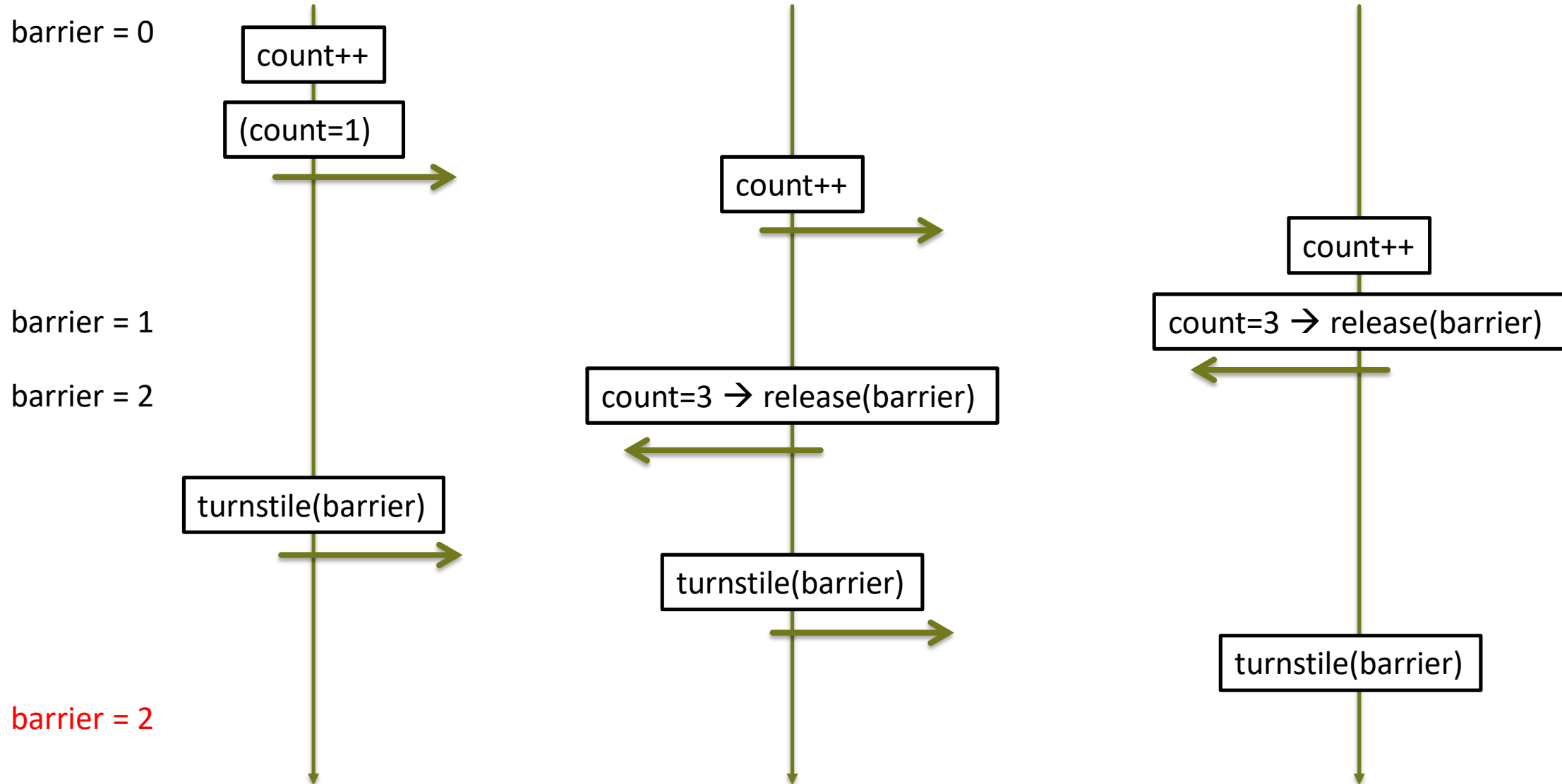
«Only when all processes have reached the turnstyle it will be opened the first time"

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

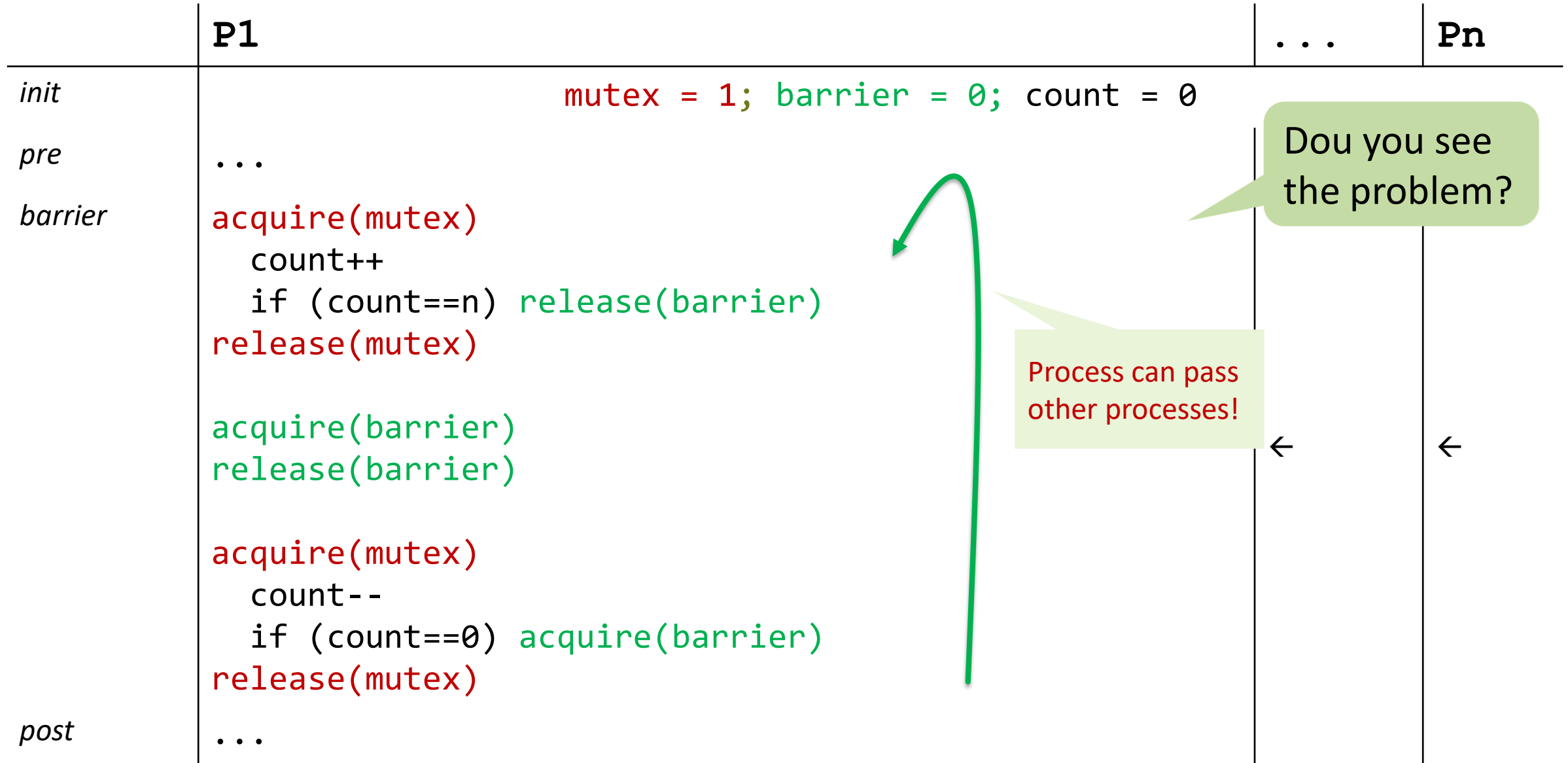
«When all processes have run through the barrier then barrier = 0" (violated)



Illustration of the problem: scheduling scenario



Reusable Barrier. 2nd trial.



Reusable Barrier. 2nd trial.

	P1	...	Pn
<i>init</i>	<code>mutex = 1; barrier = 0; count = 0</code>		
<i>pre</i>	...		
<i>barrier</i>	<code>acquire(mutex)</code> <code>count++</code> <code>if (count==n) release(barrier)</code> <code>release(mutex)</code> <code>acquire(barrier)</code> <code>release(barrier)</code> <code>acquire(mutex)</code> <code>count--</code> <code>if (count==0) acquire(barrier)</code> <code>release(mutex)</code>		
<i>post</i>	...		

Invariants

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

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

Solution: Two-Phase Barrier

init

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

barrier

```
acquire(mutex)
```

```
count++;
```

```
if (count==n)
```

```
    acquire(barrier2); release(barrier1)
```

```
release(mutex)
```

```
acquire(barrier1); release(barrier1);
```

```
// barrier1 = 1 for all processes, barrier2 = 0 for all processes
```

```
acquire(mutex)
```

```
count--;
```

```
if (count==0)
```

```
    acquire(barrier1); release(barrier2)
```

```
signal(mutex)
```

```
acquire(barrier2); release(barrier2)
```

```
// barrier2 = 1 for all processes, barrier1 = 0 for all processes
```

Lesson Learned ?

- Semaphore, Rendezvous 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

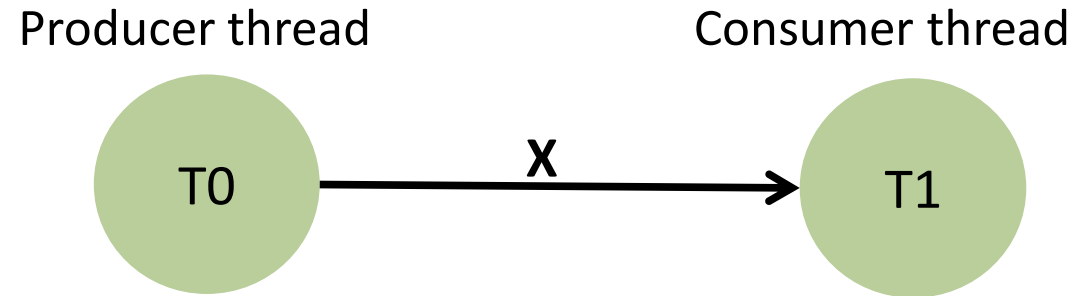
Next:

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?

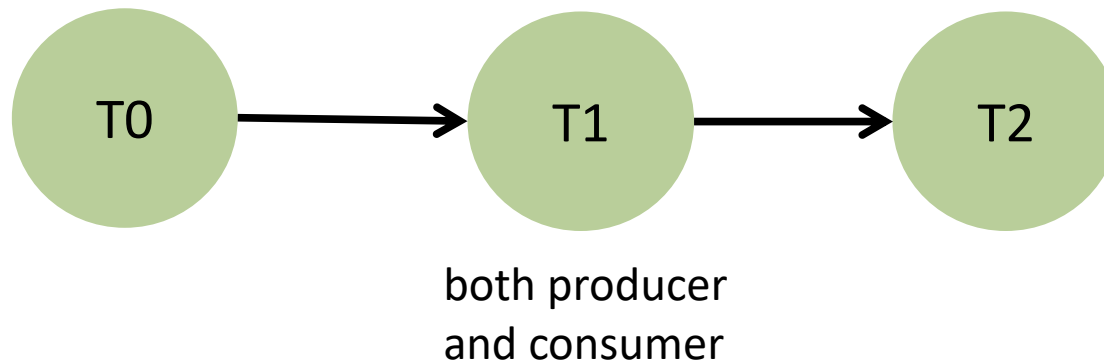
No because, at any point in time only one thread accesses X
we, however, need a synchronized mechanism to pass X from T0 to T1

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!



Analyzing tweets using Cloud Dataflow pipeline templates

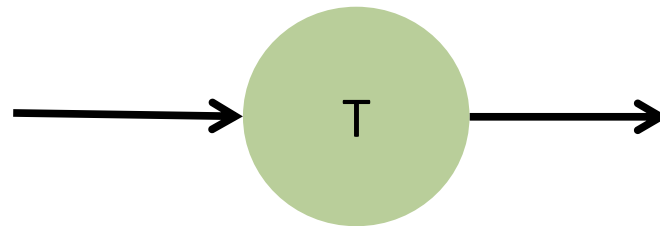
Wednesday, December 6, 2017

By Amy Unruh, Developer Relations Engineer

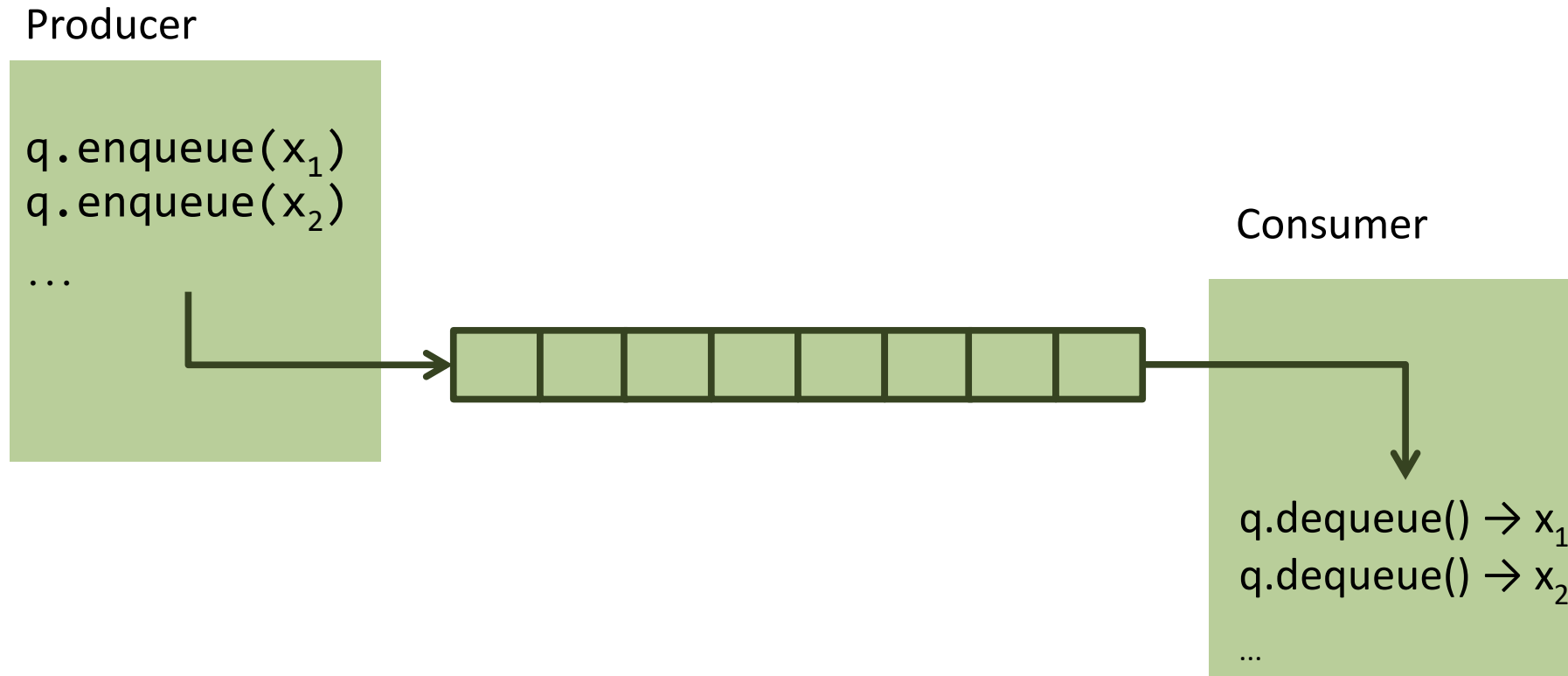
This post describes how to use Google [Cloud Dataflow templates](#) to easily launch [Dataflow](#) pipelines from a [Google App Engine \(GAE\)](#) app, in order to support [MapReduce](#) jobs and many other data processing and analysis tasks.

Pipeline Node

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

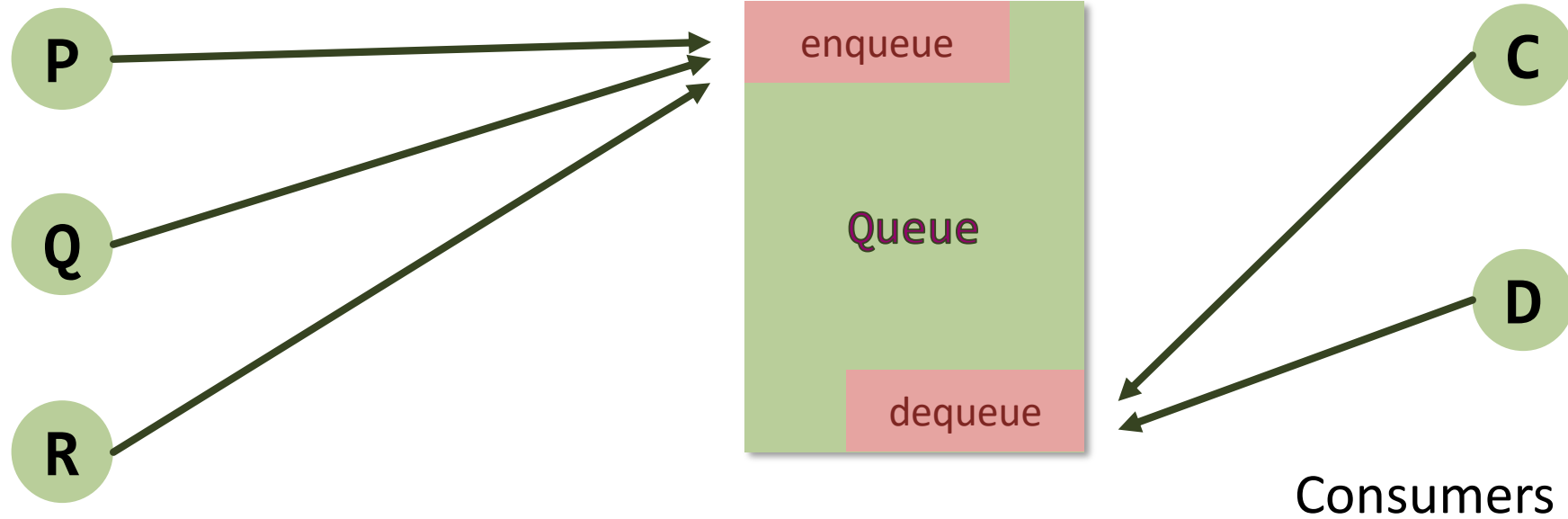


Producer / Consumer queues



Multiple Producers and Consumers

Producers

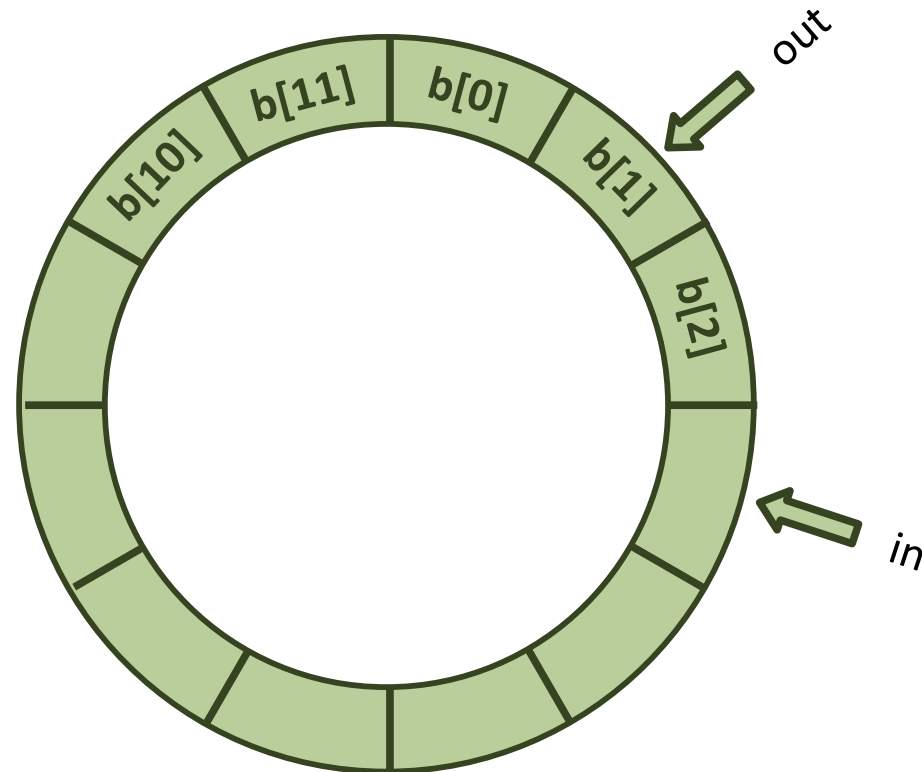


Bounded FIFO as Circular Buffer

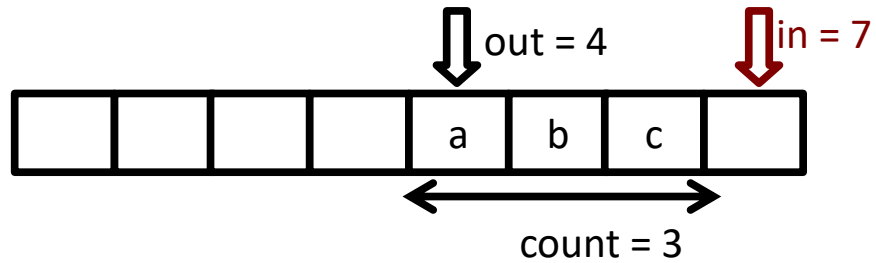


+ wrap around semantics

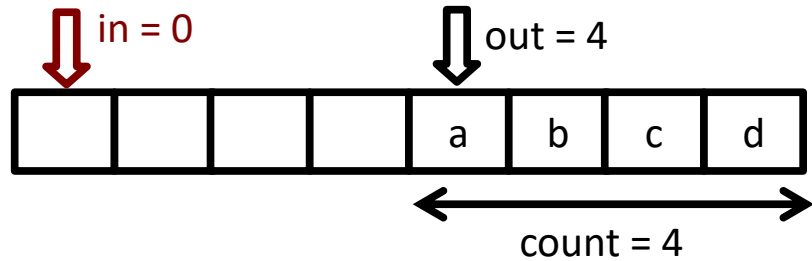
=



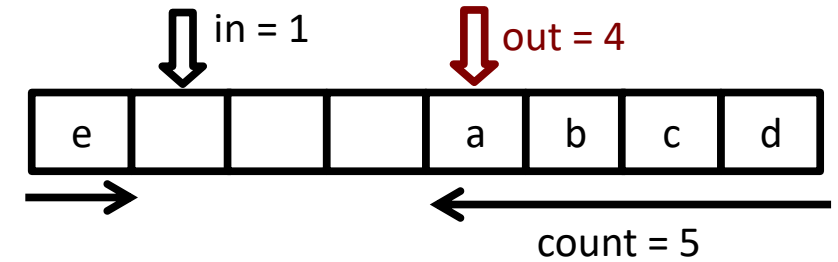
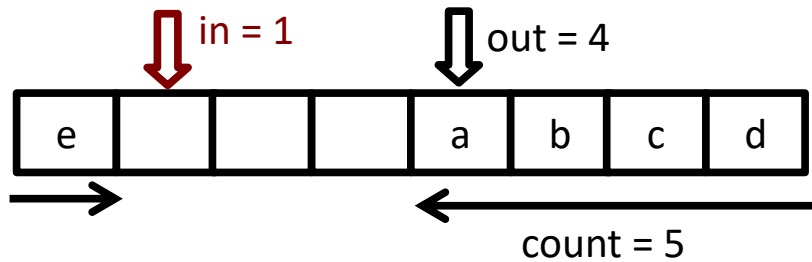
Producer / Consumer queue implementation



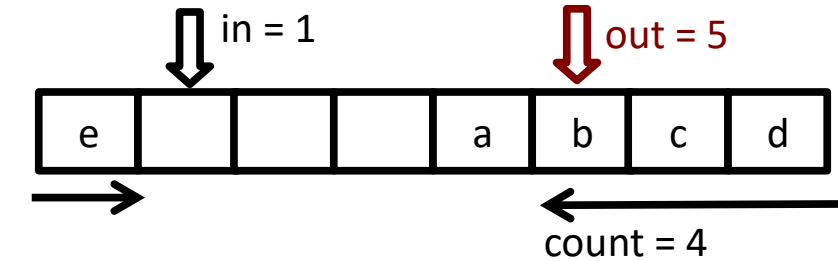
Enqueue d



Enqueue e



Dequeue → a



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?

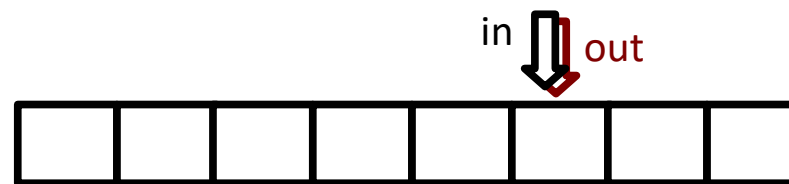
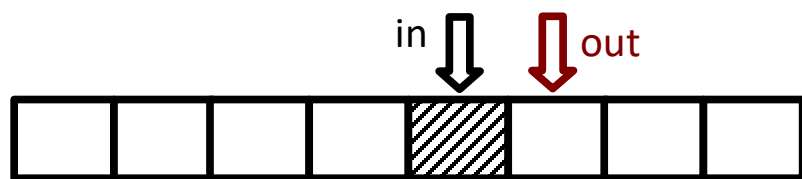
Producer / Consumer queues: helper functions

```
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
    }
}
```

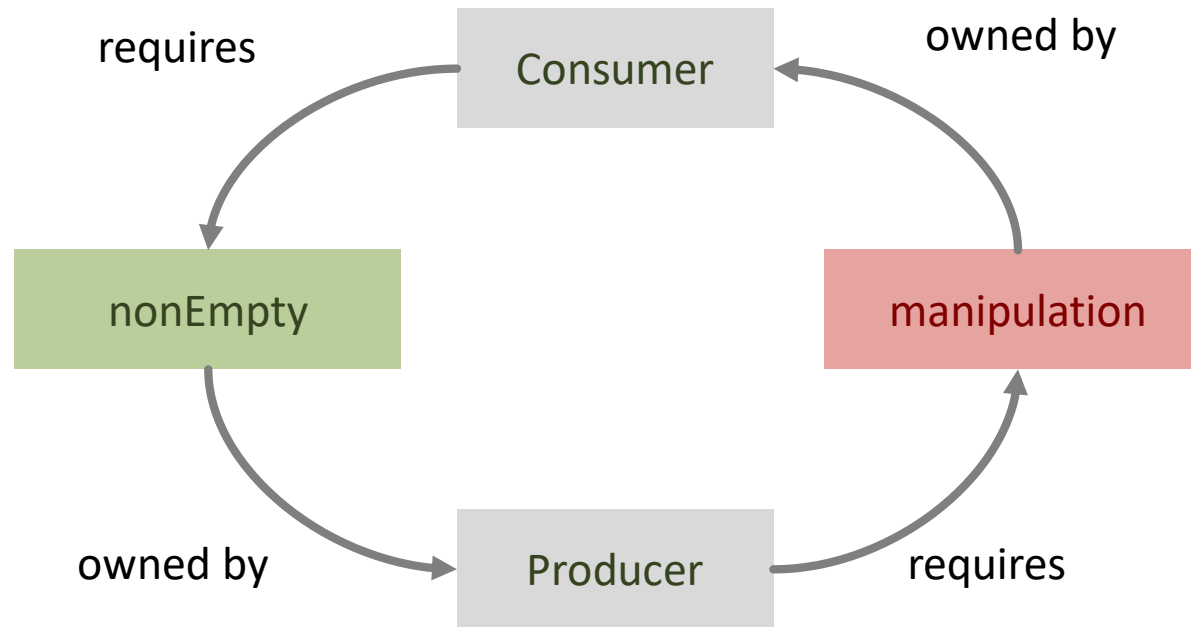
Producer / Consumer queues with semaphores, correct?

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!



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 insufficient?

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)

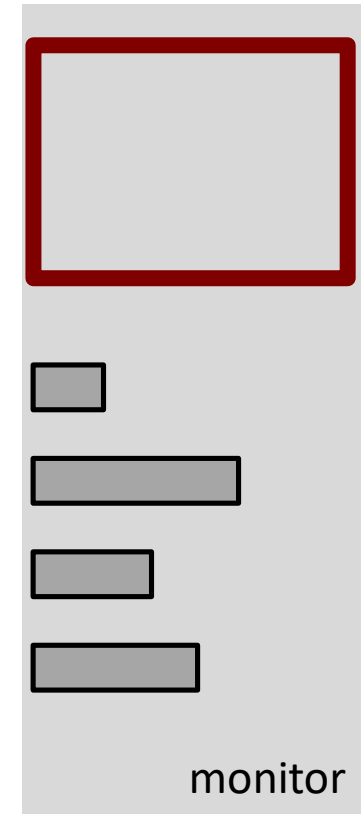
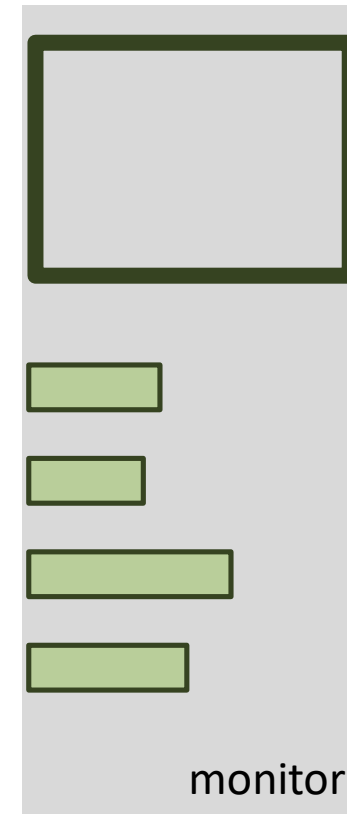
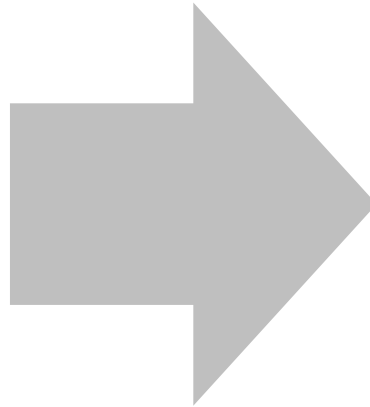
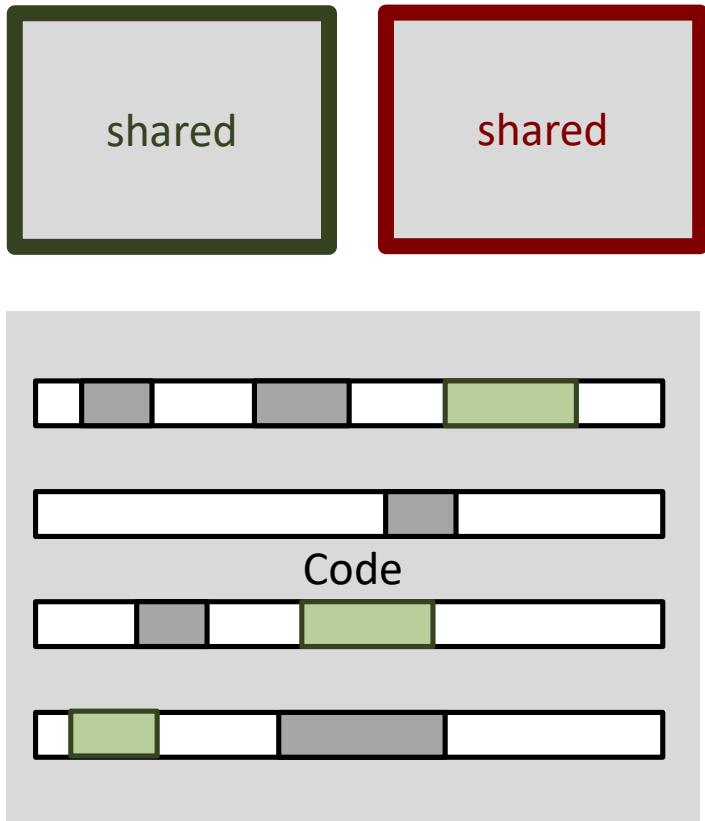


Tony Hoare
(1934-today)



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

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()?

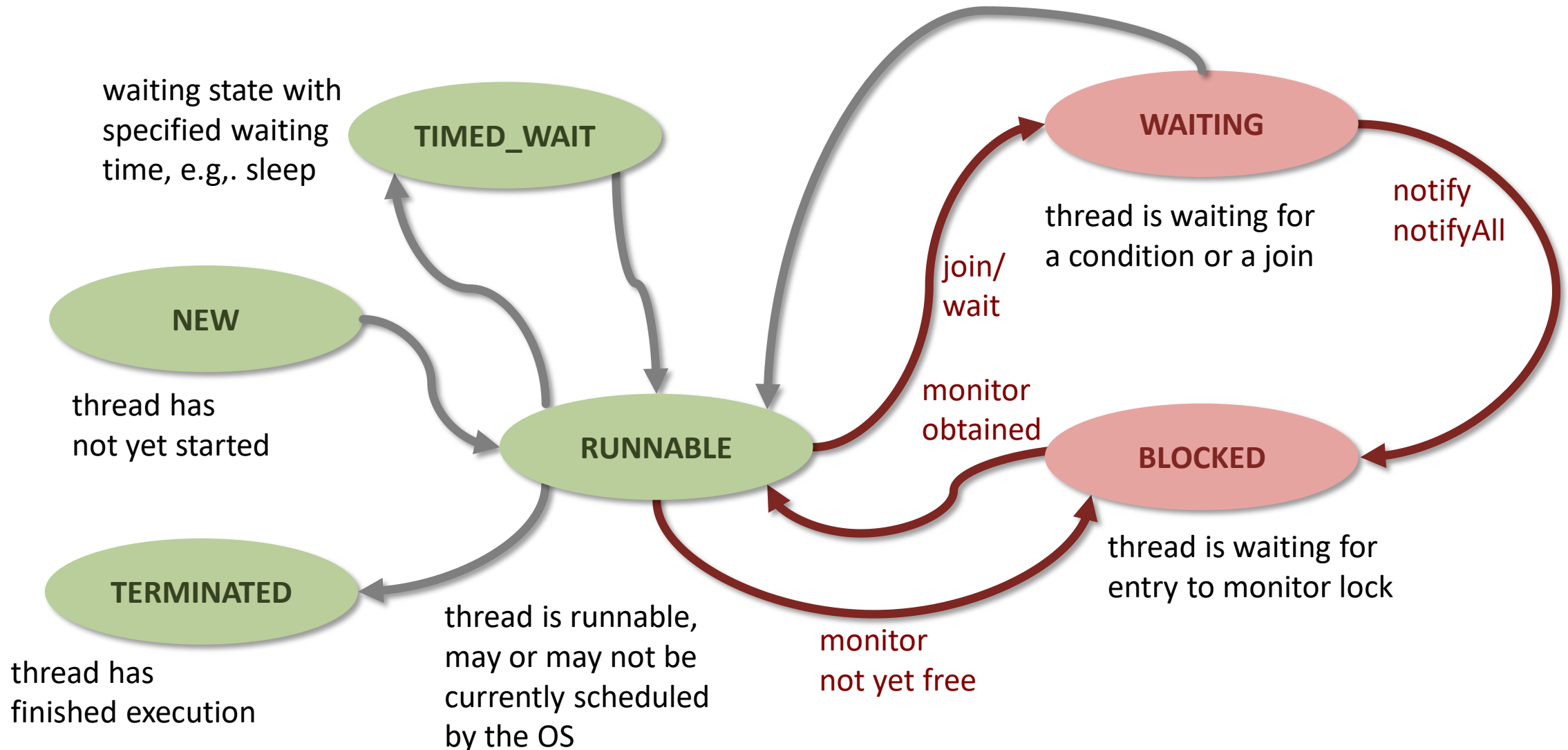
IMPORTANT TO KNOW JAVA MONITOR IMPLEMENTATION DETAILS

Thread States in Java

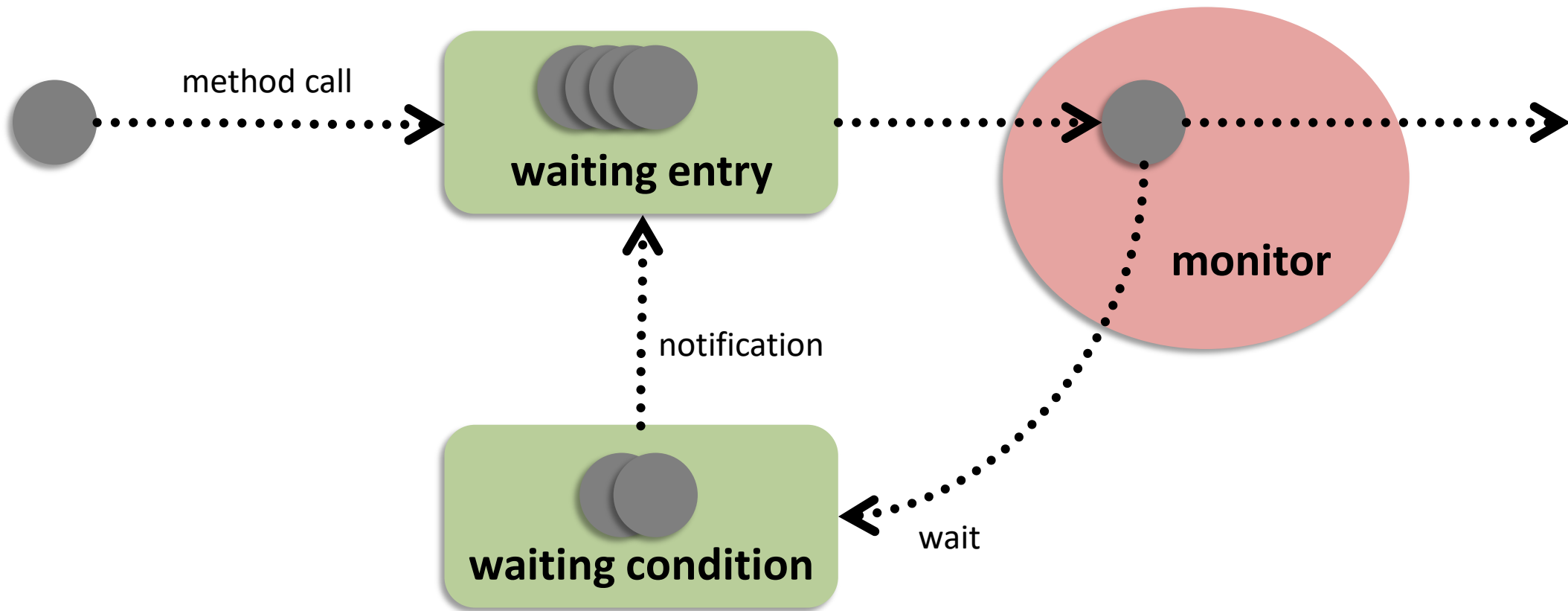
Spaced repetition

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]



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

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

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

```
synchronized void exit() {
    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;
}
```

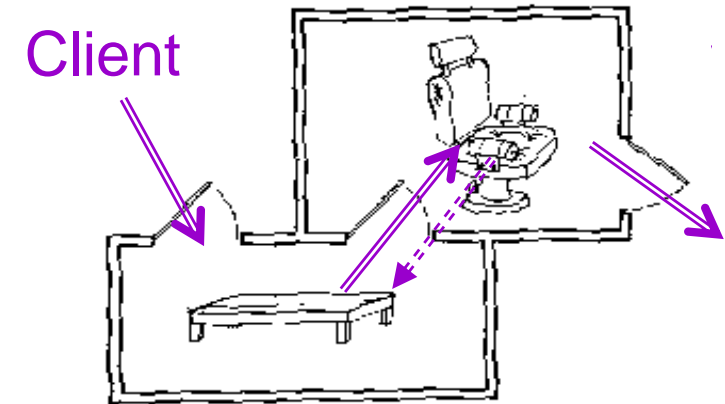
The Sleeping Barber Variant (E. Dijkstra)

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. slide 27

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, ...)**