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

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

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

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

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

- **Learning goals:**
  - Understand pitfalls in very simple synchronization algorithms
  - This is very important to design correct parallel codes
Mutual Exclusion for $n$ processes: Bakery Algorithm (1974)

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

CS Entry: Wait until ticket number is lowest
Bakery algorithm (two processes, simplified)

volatile int np = 0, nq = 0

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

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

Q also wants access

and Q has an earlier ticket

np == nq can happen
➔ global ordering of processes
Bakery algorithm (n processes)

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

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

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

(k, l_k) < (j, l_j) ⇔ l_k < l_j or (l_k = l_j and k < j)
Bakery Lock in Java

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

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

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

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

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

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

Shared memory locations (atomic registers) come in different variants

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

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

---

**Theorem 5.1 in [1]**: Bounds on Shared Memory for Mutual Exclusion

*James E. Burns*

*Georgia Institute of Technology, Atlanta, Georgia 30332*

*And*

*Nancy A. Lynch*

*Massachusetts Institute of Technology, Cambridge, Massachusetts 02139*

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

I and 10,000,000 threads!
We have constructed something ...

... that may not quite fulfil its purpose!

AND we cannot do better, can we?

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

- But we proved that we cannot do better. What now!?
  - Change (extend) the model with architecture engineering!
    - *modern multiprocessor architectures provide special instructions for atomically reading and writing at once!*
Hardware Support for Parallelism
Read-Modify-Write Operations
Hardware support for atomic operations: Example (x86)

**CMPXCHG**

Compare and Exchange

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

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

When the first operand is a memory operand, the CMPXCHG instruction compares the value in that memory location with the value in the second operand and updates the flags as above. The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically.

The LOCK prefix is used to ensure that memory operations are performed atomically, meaning that they are executed without interruption by other processes. It is used to prevent race conditions and ensure data consistency.

**Related Instructions**

CMPXCHG8B, CMPXCHG16B

From the AMD64 Architecture Programmer’s Manual

R. Hudson: IA memory ordering: [https://www.youtube.com/watch?v=WUfvvFD5tAA](https://www.youtube.com/watch?v=WUfvvFD5tAA) (2008)
Hardware support for atomic operations: Example (ARM)

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

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

From the ARM Architecture Reference Manual
Hardware support for atomic operations

Typical instructions

Test-And-Set (TAS)
   Example TSL register, flag (Motorola 68000)

Compare-And-Swap (CAS)
   Example: LOCK CMPXCHG (Intel x86)
   Example: CASA (Sparc)

Load Linked / Store Conditional
   Example LDREX/STREX (ARM)
   Example LL / SC (MIPS, POWER, RISC V)

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

[1]: H. Schweizer, M. Besta, T. Hoefler: Evaluating the Cost of Atomic Operations on Modern Architectures, ACM PACT’15
### Semantics of TAS and CAS

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

```java
int CAS (memref a, int old, int new) {
    oldval = mem[a];
    if (old == oldval) {
        mem[a] = new;
        return oldval;
    } else
    return false;
}
```

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

### Test and Set (TAS)

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

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

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

### Compare and Swap (CAS)

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

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

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

- **ignore result**
Read-Modify-Write in Java
Let's try it.

Need support for atomic operations on a high level.

Available in Java (from JDK 5) with class

```
java.util.concurrent.atomic.AtomicBoolean
```

Operations

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

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

sets `newValue` and returns previous value.
How does this work?

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

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

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

**Parameters:**
- `expect` the expected value
- `update` the new value

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

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

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

    public void lock() {
        while(state.getAndSet(true)) {} // Keep trying until the lock is acquired (return value is false).
    }

    public void unlock() {
        state.set(false); // Release the lock (set to 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

TAS

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

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

Test-and-Test-and-Set (TATAS) Lock
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  
scott@cs.wfu.edu  
Dept. of Computer Science  
Wfu, U., St. Louis

Tim Harrison  
harrison@cs.wfu.edu  
Dept. of Computer Science  
Wfu, U., St. Louis


**Abstract**

This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multithreading 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 when “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.

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

**Definition:** A Singleton is straightforward.

```java
class Singleton
{
    // static Singleton instance;
    static Singleton instance;

    public:
    static Singleton *getInstance()
    {
        if (instance == nil) // Critical section:
            instance = new Singleton;
        return instance;
    }
};
```

Problem: Memory ordering leads to race-conditions!
TATAS with backoff

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

Solution
- threads go to sleep with random duration
- increase expected duration each time the resource is not free
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) {} } // catch (InterruptedException ex) {}
    }
}
exponential backoff

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

TAS
TTAS
BackoffLock

ms/thread

number threads
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);
    }
}
Deadlocks – Motivation

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

Thread 1: $x$.transferTo(1, $y$)
- acquire lock for $x$
- withdraw from $x$

Thread 2: $y$.transferTo(1, $x$)
- acquire lock for $y$
- withdraw from $y$
- acquire lock for $x$

```java
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);
    }
}
```
Deadlocks

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

Graphically: Threads and Resources (Locks)

Thread P *attempts to* acquire resource a:

Resource b is *held by* thread Q:
Deadlocks – more formally

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

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

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

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

- two-phase locking with retry (release when failed)
  - Usually in databases where transactions can be aborted without consequence
- resource ordering
  - Usually in parallel programming where global state is modified
Back to our example: what can we do?

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

Option 1: non-overlapping (smaller) critical sections

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

```java
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"
    }
}
```

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);`

*Do you find the two problems?*

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)
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. 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 $s \geq 0$ and the following operations:

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

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

* Dijkstra called them $P$ (probeeren), $V$ (vrijgeven), also often used: *wait and signal*
Building a lock with a semaphore

mutex = Semaphore(1);

lock mutex := mutex.acquire()
    only one thread is allowed into the critical section

unlock mutex := mutex.release()
    one other thread will be let in

Semaphore number:
    1 → unlocked
    0 → locked
    x>0 → x threads will be let into “critical section”
Example: scaled dot product

- Execute in parallel: \( x = (a^T \cdot d) \cdot z \)
  - \( a \) and \( d \) are column vectors
  - \( x, z \) are scalar
- Assume each vector has 4 elements
  - \( x = (a_1 \cdot d_1 + a_2 \cdot d_2 + a_3 \cdot d_3 + a_4 \cdot d_4) \cdot z \)
- Parallelize on two processors (using two threads A and B)
  - \( x_A = a_1 \cdot d_1 + a_2 \cdot d_2 \)
  - \( x_B = a_3 \cdot d_3 + a_4 \cdot d_4 \)
  - \( x = (x_A + x_B) \cdot z \)
- 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**

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td><strong>P</strong>_<strong>Arrived</strong>=0</td>
<td><strong>Q</strong>_<strong>Arrived</strong>=0</td>
</tr>
<tr>
<td><strong>pre</strong></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>rendezvous</strong></td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td><strong>post</strong></td>
<td>...</td>
<td>..</td>
</tr>
</tbody>
</table>
Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous) Semaphores \texttt{P\_Arrived} and \texttt{Q\_Arrived}

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{init}</td>
<td>\texttt{P_Arrived=0}</td>
<td>\texttt{Q_Arrived=0}</td>
</tr>
<tr>
<td>\textit{pre}</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>\textit{rendezvous}</td>
<td>\texttt{release(P_Arrived)} ?</td>
<td>\texttt{acquire(P_Arrived)} ?</td>
</tr>
<tr>
<td>\textit{post}</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)
Semaphores \texttt{P\_Arrived} and \texttt{Q\_Arrived}

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{init}</td>
<td>\textit{pre}</td>
</tr>
<tr>
<td>$\text{P_Arrived}=0$</td>
<td>$\text{Q_Arrived}=0$</td>
</tr>
<tr>
<td>\textit{rendezvous}</td>
<td>\textit{rendezvous}</td>
</tr>
<tr>
<td>$\text{acquire(}\text{Q_Arrived})$</td>
<td>$\text{acquire(}\text{P_Arrived})$</td>
</tr>
<tr>
<td>$\text{release(}\text{P_Arrived})$</td>
<td>$\text{release(}\text{Q_Arrived})$</td>
</tr>
<tr>
<td>\textit{post}</td>
<td></td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>
Deadlock

init: P_Arrived=0
pre: ...
rendezvous: acquire(Q_Arrived)
          release(P_Arrived)
post: ...

Q_Arrived=0
pre: ...
rendezvous: acquire(P_Arrived)
          release(Q_Arrived)
post: ...
Rendezvous with Semaphores
Wrong solution with Deadlock

P
pre
acquire
release

Q
pre
acquire
release
Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)
Assume Semaphores \texttt{P\_Arrived} and \texttt{Q\_Arrived}

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{init}</td>
<td>\texttt{P_Arrived=0}</td>
<td>\texttt{Q_Arrived=0}</td>
</tr>
<tr>
<td>\textit{pre}</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>\textit{rendezvous}</td>
<td>\texttt{release(P_Arrived)}</td>
<td>\texttt{acquire(P_Arrived)}</td>
</tr>
<tr>
<td></td>
<td>\texttt{acquire(Q_Arrived)}</td>
<td></td>
</tr>
<tr>
<td>\textit{post}</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Implementing Semaphores without Spinning (blocking queues)

Consider a process list \( Q_s \) associated with semaphore \( S \)

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

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

**P first**

- **P**
  - pre
  - release
  - acquire
  - post

- **Q**
  - pre
  - acquire
  - release
  - post

**Q first**

- **P**
  - pre
  - release
  - acquire
  - post

- **Q**
  - pre
  - acquire
  - release
  - post

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>P_Arrived=0</td>
<td>Q_Arrived=0</td>
</tr>
<tr>
<td><strong>pre</strong></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>rendezvous</strong></td>
<td>release(P_Arrived)</td>
<td>acquire(P_Arrived)</td>
</tr>
<tr>
<td></td>
<td>acquire(Q_Arrived)</td>
<td>release(Q_Arrived)</td>
</tr>
<tr>
<td><strong>post</strong></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

release signals (arrow)
acquire may wait (filled box)
Rendezvous with Semaphores

Synchronize Processes P and Q at one location (Rendezvous)

Assume Semaphores $P_{\text{Arrived}}$ and $Q_{\text{Arrived}}$

<table>
<thead>
<tr>
<th></th>
<th>$P$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>$P_{\text{Arrived}}=0$</td>
<td>$Q_{\text{Arrived}}=0$</td>
</tr>
<tr>
<td>pre</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>rendezvous</td>
<td>release($P_{\text{Arrived}}$)</td>
<td>release($Q_{\text{Arrived}}$)</td>
</tr>
<tr>
<td></td>
<td>acquire($Q_{\text{Arrived}}$)</td>
<td>acquire($P_{\text{Arrived}}$)</td>
</tr>
<tr>
<td>post</td>
<td>...</td>
<td>..</td>
</tr>
</tbody>
</table>
That’s even better.

**P first**

- **P**: 
  - pre
  - release
  - acquire
  - post

- **Q**: 
  - pre
  - release
  - acquire
  - post

**Q first**

- **P**: 
  - pre
  - release
  - acquire
  - post

- **Q**: 
  - pre
  - release
  - acquire
  - post

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>P_Arrived=0</td>
<td>Q_Arrived=0</td>
</tr>
<tr>
<td>pre</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>rendezvous</td>
<td>release(P_Arrived)</td>
<td>release(Q_Arrived)</td>
</tr>
<tr>
<td></td>
<td>acquire(Q_Arrived)</td>
<td>acquire(P_Arrived)</td>
</tr>
<tr>
<td>post</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

release signals (arrow) 
acquire may wait (filled box)
Back to our dot-product

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

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

Synchronize a number of processes.

How would you implement this using Semaphores?
### Barrier – 1\textsuperscript{st} try

Synchronize a number (n) of processes. Semaphore \texttt{barrier}. Integer \texttt{count}.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>...</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{init}</td>
<td>\texttt{barrier} = 0; volatile \texttt{count} = 0</td>
<td>\texttt{Race Condition !}</td>
<td></td>
</tr>
<tr>
<td>\textit{pre}</td>
<td>...</td>
<td>\texttt{Some wait forever!}</td>
<td></td>
</tr>
<tr>
<td>\textit{barrier}</td>
<td>\texttt{count}++</td>
<td>\texttt{←}</td>
<td>\texttt{←}</td>
</tr>
<tr>
<td>\textit{if (count==n)} \texttt{release(barrier)}</td>
<td>\texttt{←}</td>
<td>\texttt{←}</td>
<td>\texttt{←}</td>
</tr>
<tr>
<td>\texttt{acquire(barrier)}</td>
<td>\texttt{←}</td>
<td>\texttt{←}</td>
<td>\texttt{←}</td>
</tr>
<tr>
<td>\textit{post}</td>
<td>...</td>
<td>\texttt{←}</td>
<td>\texttt{←}</td>
</tr>
</tbody>
</table>
Barrier

Synchronize a number (n) of processes.
Semaphore **barrier**. Integer count.

<table>
<thead>
<tr>
<th>init</th>
<th>barrier = 0; volatile count</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>...</td>
</tr>
<tr>
<td>barrier</td>
<td>count++</td>
</tr>
<tr>
<td></td>
<td>if (count==n) release(barrier)</td>
</tr>
<tr>
<td></td>
<td>acquire(barrier)</td>
</tr>
<tr>
<td>post</td>
<td>...</td>
</tr>
</tbody>
</table>

**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

**Process P**
- `x++`
- `reg = x`
- `reg = reg + 1`
- `x = reg`
- `read x`
- `write x`

**Process Q**
- `write x`
- `read x`
- `reg = x`
- `reg = reg - 1`
- `x = reg`
- `write x`

**Race Condition**
With Mutual Exclusion

Process P

Critical Section

reg = x
reg = reg + 1
x = reg

read x
write x

Mutual Exclusion

Process Q

Critical Section

reg = x
reg = reg - 1
x = reg

read x
write x

x++

x--
Barrier

Synchronize a number (n) of processes. Semaphores `barrier`, `mutex`. Integer count.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>...</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td><code>mutex = 1; barrier = 0; count = 0</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pre</strong></td>
<td><code>...</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| **barrier** | `acquire(mutex)` | | `←` `←` `←`
| | `count++` | | |
| | `release(mutex)` | | |
| | `if (count==n) release(barrier)` | | |
| | `acquire(barrier)` | | |
| | `release(barrier)` | | |
Reusuable Barrier. 1st trial.

<table>
<thead>
<tr>
<th>P1</th>
<th>...</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>mutex = 1; barrier = 0; count = 0</td>
<td></td>
</tr>
<tr>
<td><strong>pre</strong></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
| **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.

<table>
<thead>
<tr>
<th>P1</th>
<th>init</th>
<th>mutex = 1; barrier = 0; count = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pre</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>barrier</td>
<td>acquire(mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>count++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>release(mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (count==n) release(barrier)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>acquire(barrier)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>release(barrier)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>acquire(mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>count--</td>
</tr>
<tr>
<td></td>
<td></td>
<td>release(mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (count==0) acquire(barrier)</td>
</tr>
<tr>
<td></td>
<td>post</td>
<td>...</td>
</tr>
</tbody>
</table>

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

barrier = 0

count++
(count=1)

turnstile(barrier)

barrier = 1

barrier = 2

count++

count=3 \rightarrow \text{release(barrier)}

turnstile(barrier)

barrier = 2

count++

count=3 \rightarrow \text{release(barrier)}

turnstile(barrier)

barrier = 2
### Reusable Barrier. 2nd trial.

<table>
<thead>
<tr>
<th>P1</th>
<th>...</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>( \text{mutex} = 1; \text{barrier} = 0; \text{count} = 0 )</td>
<td><strong>post</strong></td>
</tr>
<tr>
<td><strong>pre</strong></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| **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) |

**Dou you see the problem?**

Process can pass other processes!
## Reusable Barrier. 2nd trial.

<table>
<thead>
<tr>
<th>P1</th>
<th>...</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>mutex = 1; barrier = 0; count = 0</td>
<td></td>
</tr>
<tr>
<td><strong>pre</strong></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
| **barrier** | acquire(mutex)  
  count++  
  if (count==n) release(barrier)  
  release(mutex) |  |
| | acquire(barrier)  
  release(barrier) |  |
| | acquire(mutex)  
  count--  
  if (count==0) acquire(barrier)  
  release(mutex) |  |
| **post** | ... |  |

### Invariants

- "When all processes have passed the barrier, it holds that barrier = 0"
- "Even when a single process has passed the barrier, it holds that barrier = 0" (violated)
Solution: Two-Phase Barrier

\[ \text{init} \]

\[ \text{mutex}=1; \quad \text{barrier}_1=0; \quad \text{barrier}_2=1; \quad \text{count}=0 \]

\[ \text{acquire(mutex)} \]
\[ \text{count}++; \]
\[ \text{if (count==n)} \]
\[ \text{acquire(barrier}_2); \quad \text{release(barrier}_1) \]

\[ \text{release(mutex)} \]

\[ \text{acquire(barrier}_1); \quad \text{release(barrier}_1); \]
\[ // \text{barrier}_1 = 1 \text{ for all processes, barrier}_2 = 0 \text{ for all processes} \]

\[ \text{acquire(mutex)} \]
\[ \text{count}--; \]
\[ \text{if (count==0)} \]
\[ \text{acquire(barrier}_1); \quad \text{release(barrier}_2) \]

\[ \text{signal(mutex)} \]

\[ \text{acquire(barrier}_2); \quad \text{release(barrier}_2) \]
\[ // \text{barrier}_2 = 1 \text{ for all processes, barrier}_1 = 0 \text{ for all processes} \]

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

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

Locks are not enough: we need methods to wait for events / notifications
Semaphores
Rendezvous and Barriers

Next lecture:
Producer-Consumer Problem
Monitors and condition variables