Motivation

Earlier solutions to the critical section problem, like Peterson's algorithm, are quite complex, error prone, and inefficient.

We need to have a better solution so that the problem can be solved in a more disciplined way.
**Semaphore**

A *semaphore* is a synchronization construct that can be used to provide **mutual exclusion** and **conditional synchronization**.

From another perspective, a *semaphore* is a shared object that can be manipulated only by two atomic operations, *P* and *V*.

---

**Counting & Binary Semaphores**

There are two types of semaphores: **Counting Semaphore** and **Binary Semaphore**.

*Counting Semaphore* can be used for mutual exclusion and conditional synchronization.

*Binary Semaphore* is specially designed for mutual exclusion.
**Counting Semaphore**

A counting semaphore can be considered as a pool of permits.

A thread uses P operation to request a permit. If the pool is empty, the thread waits until a permit becomes available.

A thread uses V operation to return a permit to the pool.

---

**An OO Definition**

A counting semaphore can be defined in an object-oriented manner as shown below:

```java
class CountingSemaphore {
    private int permits;
    public CountingSemaphore (int initialPermits) { permits = initialPermits; }
    public void P() { ... }
    public void V() { ... }
}
```
**An Implementation Sketch**

Here is a sketch of one possible implementation of methods \( P() \) and \( V() \):

```java
public void P() {
    permits = permits - 1;
    if (permits < 0) {
        wait on a queue of blocked threads;
    }
}
public void V() {
    ++ permits;
    if (permits <= 0) {
        notify one waiting thread;
    }
}
```

**Invariant**

For a counting semaphore \( s \), at any time, the following condition holds:

\[
\text{(the initial number of permits)} + \text{(the number of completed s.V operations)} \geq \text{(the number of completed s.P operations)}
\]
**Binary Semaphore**

A binary semaphore must be initialized with 1 or 0, and the completion of P and V operations must alternate.

If the semaphore is initialized with 1, then the first completed operation must be P. If the semaphore is initialized with 0, then the first completed operation must be V.

Both P and V operations can be blocked, if they are attempted in a consecutive manner.

---

**Counting vs. Binary Semaphore**

<table>
<thead>
<tr>
<th>Counting Semaphore</th>
<th>Binary Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can take any initial value</td>
<td>Can only take 0 or 1</td>
</tr>
<tr>
<td>V operation never blocks</td>
<td>Both P and V operation may block</td>
</tr>
<tr>
<td>Completed P and V operations do not have to alternate</td>
<td>Completed P and V operations must alternate</td>
</tr>
<tr>
<td>V could always be the first completed operation</td>
<td>If the initial value is 0, the first completed operation must be V, if the initial value is 1, the first completed operation must be P.</td>
</tr>
</tbody>
</table>
**Simulating Counting Semaphores**

```
public final class CountingSemaphore {
    private int permits = 0;
    BinarySemaphore mutex (1);
    BinarySemaphore delayQ (0);

    public CountingSemaphore (int initialPermits) {
        permits = initialPermits;
    }

    public void P () {
        mutex.P ();
        -- permits;
        if (permits < 0) {
            mutex.V ();
            delayQ.P ();
        } else
            mutex.V ();
    }

    public void V () {
        mutex.P ();
        ++ permits;
        if (permits <= 0) {
            delayQ.V ();
        } mutex.V ();
    }
}
```

**A scenario**

```
T1     T2     T3     T4
(1)    (1)    (7)    (7)
(2)    (2)    (8)    (8)
(3)    (3)    (9)    (9)*
(4)    (4)    (10)  (10)*
(5)    (5)  *10    *10
```
**Lock**

*Lock* is another synchronization construct that can be used to solve the critical section problem.

A *lock* defines two types of operations: *lock* and *unlock*.

---

**Lock Ownership**

A *lock* can be owned by at most one thread at any given time.

A thread that calls *lock* becomes the owner of a *lock* if the lock is not owned by any other thread; otherwise, the thread is blocked.

The owner of a lock can release the ownership by calling *unlock*.

**Important**: The owner of a lock is not blocked if it calls *lock* again. However, the owner must call *unlock* the same number of times to release the ownership.
Lock vs. Binary Semaphore

For a binary semaphore, consecutive P operations will be blocked. But a thread that owns a lock can invoke lock operations again without being blocked.

The owner for calls to lock and unlock must be the same thread. But calls to P and V can be made by different threads.

```
Lock l = new Lock();
BinarySemaphore s = new BinarySemaphore(1);
```

```
T1:
  l.lock();
  l.lock();
  ...

T1:
  s.P();
  s.P();
  ...

T1:
  l.lock();
  T2:
  l.unlock();
  T2:
  s.V();
```
**synchronized method**

Each object has a lock.

```java
public synchronized void foo () {
    this.lock ();
    ...
    bar ();
    ...
    this.unlock ();
}
```

Each object has a BinarySemaphore.

```java
public synchronized void foo () {
    this.P ();
    ...
    bar ();
    ...
    this.V ();
}
```

```
public synchronized void bar () {
    this.lock ();
    ...
    this.unlock ();
}
```

```
public synchronized void bar () {
    this.P ();
    ...
    this.V ();
}
```

Each object has a lock.

```
public synchronized void foo () {
    this.lock ();
    ...
    bar ();
    ...
    this.unlock ();
}
```

```
public synchronized void foo () {
    this.P ();
    ...
    bar ();
    ...
    this.V ();
}
```

```
public synchronized void bar () {
    this.lock ();
    ...
    this.unlock ();
}
```

```
public synchronized void bar () {
    this.P ();
    ...
    this.V ();
}
```

---

**Binary Semaphore vs. Lock**

**Binary Semaphore**
- Has no concept of ownership
- Any thread can invoke P or V operations
- Consecutive P (or V) operations will be blocked
- Need to specify an initial value

**Lock**
- A lock can be owned by at most one thread at any given time
- Only the owner can invoke unlock operations
- The owner can invoke lock (or unlock) operations in a row.
- Does not have to be initialized
**Semaphore**

- Introduction
- Semaphore Implementation
- Semaphore in Java
- Semaphore-based Solutions

---

**Implementation 1**

```java
public void P() {
    while (permits == 0) {
        sleep (interval);
    }
    permits = permits - 1;
}

public void V() {
    permits = permits + 1;
}
```
**Implementation 2**

```java
public void P() {
    permits = permits - 1;
    if (permits < 0) {
        wait on a queue of blocked threads;
    }
}

public void V() {
    permits = permits + 1;
    if (permits == 0) {
        notify one waiting thread;
    }
}
```

**Implementation 3**

```java
public void P() {
    if (permits == 0) {
        wait on a queue of blocked threads;
    }
    permits = 0;
    if (queue of threads blocked in V() is not empty) {
        awaken one waiting thread in V();
    }
}

public void V() {
    if (permits == 1) {
        wait on a queue of blocked threads;
    }
    permits = 1;
    if (queue of threads blocked in P() is not empty) {
        awaken one waiting thread in P();
    }
}
```
**Mutual exclusion for permits**

```java
public void P() {
    entry-section;
    permits = permits - 1;
    if (permits < 0) {
        exit-section;
        wait on a queue of blocked threads;
        entry-section;
    }
    exit-section;
}

class permits {
    private int permits;
    public void P() {
        entry-section;
        permits = permits - 1;
        if (permits < 0) {
            exit-section;
            wait on a queue of blocked threads;
            entry-section;
        }
        exit-section;
    }

    public void V() {
        entry-section;
        permits = permits + 1;
        if (permits <= 0) { 
            notify one waiting thread;
        }
        exit-section;
    }
}
```

**VP Operation**

Let $s$ and $t$ be two semaphores.

An execution of $t.VP(s)$ is equivalent to $s.V(); t.P();$ except that during the execution of $t.VP(s)$, no intervening $P(), V()$ or $VP()$ operations are allowed to be completed on $s$ and $t$. 
VP Operation (Cont’d)

Consider the following two program fragments:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.V()</td>
<td>t.P()</td>
</tr>
<tr>
<td>t.P()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t.VP(s)</td>
<td>t.P()</td>
</tr>
</tbody>
</table>

Semaphore

- Introduction
- Semaphore Implementation
- Semaphore in Java
- Semaphore-based Solutions
**Abstract Definition**

Java does not provide any semaphore classes. We can however simulate semaphores in Java.

```java
public abstract class Semaphore {
    protected int permits;
    protected abstract void P();
    protected abstract void V();
    protected Semaphore (int initialPermits) { permits = initialPermits; }
}
```

**CountingSemaphore**

```java
public final class CountingSemaphore extends Semaphore {
    public CountingSemaphore (int initialPermits) {
        super(initialPermits);
    }
    synchronized public void P () {
        permits --;
        if (permits < 0) {
            try { wait(); } catch (InterruptedException ex) {};
        }
    }
    synchronized public void V () {
        ++ permits;
        if (permits <= 0) {
            notify();
        }
    }
}
```
BinarySemaphore (cont’d)

If we replace "while" ... "notifyAll" with "if" ... "notify", does it still implement BinarySemaphore correctly?
**MutexLock**

```java
public final class MutexLock {
    private Thread owner = null;
    private int waiting = 0;
    public int count = 0;
    public boolean free = true;

    public synchronized void lock () {
        if (free) {
            count = 1; free = false; owner = Thread.currentThread();
        } else if (owner == Thread.currentThread()) { ++ count; }
        else { ++ waiting; try { wait(); } catch (InterruptedException ex) {} } 
        free = false; count = 1; owner = Thread.currentThread();
    }

    public synchronized void unlock () {
        if (owner != null) {
            if (owner == Thread.currentThread()) {
                if (count == 0) {
                    owner = null;
                    if (waiting > 0) {
                        -- waiting;
                        free = true;
                        notify();
                    } else { free = true; return; }
                } else { free = true; return; }
            } 
            else return;
        } else throw new OwnerException();
    }
}
```

---

**Semaphore**

- **Introduction**
- **Semaphore Implementation**
- **Semaphore in Java**
- **Semaphore-based Solutions**
The CS Problem

The following simple solution applies to the general n-process CS problem.

```java
BinarySemaphore mutex = new BinarySemaphore(1);
while (true) {
    mutex.P();
    critical section
    mutex.V();
    non-critical section
}
```

Resource Allocation (1)

Consider there are three threads that contend for two resources:

```java
CountingSemaphore resources = new CountingSemaphore(2);

Thread 1
resources.P(); /* use the resource */
resources.V();

Thread 2
resources.P(); /* use the resource */
resources.V();

Thread 3
resources.P(); /* use the resource */
resources.V();
```
**Resource Allocation (2)**

Many problems require bookkeeping to be done outside of P and V methods.

```java
int count = 2;
int waiting = 0;
CountingSemaphore mutex = new CountingSemaphore(1);
CountingSemaphore resAvail = new CountingSemaphore(0);

/* before using a resource */
mutex.P();
if (count > 0) {
    count --;
    mutex.V();
} else {
    waiting ++;
    mutex.V();
    resAvail.P();
}

/* after using a resource */
mutex.P();
if (waiting > 0) {
    -- waiting;
    resAvail.V();
} else {
    count ++;
    mutex.V();
}
```

**Semaphore Patterns**

- **Mutex**: A binary semaphore or a counting semaphore initialized as 1.
- **Enter-and-Test**: A thread needs to enter a critical section before testing a condition that involves shared variables.
- **Exit-before-Wait**: A thread inside a critical section needs to release its mutual exclusion before it waits on a condition.
- **Condition queue**: A semaphore can be used as a queue of blocked threads that are waiting for a condition to become true.
The Bounded Buffer Problem

There is a single producer and a single consumer; and there is a \( n \)-slot communication buffer.

The producer deposits items into the buffer; the consumer fetches items from the buffer.

A solution to this problem should not overwrite any item and/or fetch any item more than once.

In addition, the solution should allow maximum concurrency.

Solution 1

```java
Item[] buf = new Item[n];
CountingSemaphore full = new CountingSemaphore(0);
CountingSemaphore empty = new CountingSemaphore(n);

Producer {
    int in = 0;
    Item item;
    ...
    empty.P ();
    (1)
    buf[in] = item;
    (2)
    in = (in + 1) % n;
    (3)
    full.V ();
    (4)
    ...
}

Consumer {
    int out = 0;
    Item item;
    ...
    full.P ();
    (5)
    item = buf[out];
    (6)
    out = (out + 1) % n;
    (7)
    empty.V ();
    (8)
    ...
}
```
The Bounded Buffer Problem (Cont'd)

Does solution 1 work if we have multiple producers and consumers?

Solution 2

```java
Item buf[] = new Item[n];
CountingSemaphore full = new CountingSemaphore(0);
CountingSemaphore empty = new CountingSemaphore(n);
BinarySemaphore pMutex = new BinarySemaphore(1);
BinarySemaphore cMutex = new BinarySemaphore(1);

Producer {
    int in = 0;
    Item item;
    ...
    empty.P();
    pMutex.P();
    buf[in] = item;
    in = (in + 1) % n;
    pMutex.V();
    full.V();
    ...
}

Consumer {
    int out = 0;
    Item item;
    ...
    full.P();
    cMutex.P();
    item = buf[out];
    out = (out + 1) % n;
    cMutex.V();
    full.V();
    ...
}
```
**Dining Philosophers**

Requirements:
- Absence of deadlock
- Absence of starvation
- Maximal Parallelism

---

**Solution 1**

```
BinarySemaphore chopsticks = new BinarySemaphore [n];
// initialization
for (int j = 0; j < n; j++) {
    chopsticks[j] = new BinarySemaphore (1);
}

philosopher i

while (true) {
    /* think */
    chopsticks[i].P(); (1)
    chopsticks[(i + 1) % n].P(); (2)
    /* eat */
    chopsticks[i].V(); (3)
    chopsticks[(i + 1) % n].V(); (4)
}
```
**Solution 2**

This solution is the same as solution 1 except that only \((n - 1)\) philosophers are allowed to sit at a table that has \(n\) seats.

**Solution 3**

This solution is the same as solution 1 except that one philosopher is designated as the "odd" philosopher and this odd philosopher picks up her right fork, instead of her left fork, first.
Solution 4

```c
const int thinking = 0; const int hungry = 1; const int eating = 2;
BinarySemaphore mutex(1);
int state[] = new int [n];
BinarySemaphore self[] = new BinarySemaphore [n];
for (int j = 0; j < n; j++) {
    state[j] = thinking; self[j] = new BinarySemaphore (0);
}

philosopher i
while (true) {
    /* think */
    mutex.P();
    state[i] = hungry;
    test(i);
    self[i].P();
    /* eat */
    mutex.P();
    state[i] = thinking;
    test((i - 1) % n);
    test((i + 1) % n);
    mutex.V();
}

void test (int k) {
    if ((state[k] = hungry) && (state[(k - 1) % n] != eating) 
        && (state[(k + 1) % n] != eating)) {
        state[k] = eating;
        self[k].V();
    }
```

Readers/Writers Problem

Readers may access shared data concurrently, but a writer always has exclusive access.

What's the difference between readers/writers problem and producers/consumers problem?
Access Strategies

In general, there are three categories of access strategies:

- **R = W**: Readers and writers have equal priority.
- **R > W**: Readers generally have a higher priority than writers.
- **R < W**: Readers generally have a lower priority than writers

Access Strategies (cont’d)

- **R = W.1**: One reader or one writer with equal priority.
- **R = W.2**: Many readers or one writer with equal priority.
- **R > W.1**: Many readers or one writer with readers having a higher priority.
- **R > W.2**: Same as R > W.1 except that when a reader arrives, if no other reader is reading or waiting, it waits until all writers that arrived earlier have finished.
- **R < W.1**: Many readers or one writer with writers having a higher priority.
- **R < W.2**: Same as R < W.1 except that when a writer arrives, if no other writer is waiting or writing, it waits until all readers that arrived earlier have finished.
Consider the following request queue. Identify the order in which the requests will be served according to different access strategies.

- **R = W** vs. **R > W** vs. **R < W**

| r1 | w1 | r2 | w2 |

**R > W.1 vs. R > W.2**

- **Writer 1:**
  - req w1
  - start w1
  - end w1

- **Writer 2:**
  - req w2

- **Writer 3:**
  - req w3

- **Reader 1:**
  - req r1
**R < W.1 vs. R < W.2**

**Writer 1:**

- req w1
- start w1
- end w1

**Reader 1:**

- req r1

**Reader 2:**

- req r2

**Writer 2:**

- req w2

---

**R > W.1**

```c
int activeReaders = 0, activeWriters = 0, waitingWriters = 0, waitingReaders = 0;
BinarySemaphore mutex;
CountingSemaphore readers_que(0), writer_que(0);

read () {
    mutex.P();
    if (activeWriters > 0) {
        waitingReaders ++;
        readers_que.VP(mutex);
    }
    activeReaders ++;
    if (waitingReaders > 0) {
        waitingReaders --;
        readers_que.V();
    } else {
        mutex.V();
    }
    /* read shared data */
    mutex.P();
    activeReaders --;
    if (activeReaders == 0 && waitingWriters > 0) {
        waitingWriters --;
        writers_que.V();
    } else {
        mutex.V();
    }
}
```

```c
write () {
    mutex.P();
    if (activeReaders > 0 || activeWriters > 0) {
        writers_que.VP(mutex);
    }
    activeWriters ++;
    mutex.V();
    /* write shared data */
    mutex.P();
    activeWriters --;
    if (waitingReaders > 0) {
        waitingReaders --;
        readers_que.V();
    } else if (waitingWriters > 0) {
        waitingWriters --;
        writers_que.V();
    } else {
        mutex.V();
    }
}
```
R > W.2

```c
int activeReaders = 0;
MutexLock mutex;
BinarySemaphore writers_r_que(1);

read () {
    mutex.lock();
    ++ activeReaders;
    if (activeReaders == 1) {
        writers_r_que.P();
    }
    mutex.unlock();
    /* read shared data */
    mutex.lock();
    -- activeReaders;
    if (activeReaders == 0) {
        writers_r_que.V();
    }
    mutex.unlock();
}

write () {
    writers_r_que.P();
    /* write shared data */
    writers_r_que.V();
}
```

R < W.2

```c
int activeReaders = 0;
int waitingOrWritingWriters = 0;
MutexLock mutex_r, mutex_w;
BinarySemaphore readers_w_que(1), writers_w_que(1);

read () {
    readers_w_que.P();
    mutex_r.lock();
    ++ activeReaders;
    if (activeReaders == 1) {
        writers_w_que.P();
    }
    mutex_r.unlock();
    readers_w_que.V();
    /* read shared data */
    mutex_r.lock();
    -- activeReaders;
    if (activeReaders == 0) {
        writers_w_que.V();
    }
    mutex_r.unlock();
}

write () {
    mutex_w.lock();
    ++ waitingOrWritingWriters;
    if (waitingOrWritingWriters == 1) {
        readers_w_que.P();
    }
    mutex_w.unlock();
    writers_w_que.P();
    /* write shared data */
    writers_w_que.V();
    mutex_w.lock();
    -- waitingOrWritingWriters;
    if (waitingOrWritingWriters == 0) {
        readers_w_que.V();
    }
    mutex_w.unlock();
}
```