Content influenced by many excellent references, see References slide for acknowledgements.
What is the desired property?

class Set {
    final Vector elems = new Vector();

    void add(Object x) {
        if (!elems.contains(x)) {
            elems.add(x);
        }
    }
}

class Vector {
    synchronized void add(Object o) { ... }
    synchronized boolean contains(Object o) { ... }
}
What is the desired property?

Q.insert(elem):
    atomic {
        while (Q.full()) {}
        // Add elem to the Q
    }

Q.remove():
    atomic {
        while (Q.empty()) {}
        // Return data from Q
    }
Implementing Synchronization Patterns

• Condition synchronization
  ```java
  while ¬condition
  // do nothing (spin)
  ```

• Mutual exclusion
  ```java
  lock: bool := false
  Lock.acquire():
    while TAS(&lock)
    // spin
  Lock.release():
    lock := false
  ```
Locks (Mutual Exclusion)

```java
public interface Lock {
    public void lock();
    public void unlock();
}

public class LockImpl implements Lock {
 ...
 ...
}

Lock mtx = new LockImpl(...);
...
mtx.lock();
try {
    ...
    // body
} finally {
    mtx.unlock();
}
```
Desired Synchronization Properties

• Mutual exclusion or safety
  Critical sections on the same lock from different threads do not overlap

• Livelock freedom
  If a lock is available, then some thread should be able to acquire it within bounded steps.
Desired Synchronization Properties

• Deadlock freedom

If some thread attempts to acquire the lock, then some thread should be able to acquire the lock

• Starvation freedom

• Every thread that acquires a lock eventually releases it
• A lock acquire request must eventually succeed within bounded steps
Classic Mutual Exclusion Algorithms
Peterson’s Algorithm

class PetersonLock {

    static volatile boolean[] flag = new boolean[2];
    static volatile int victim;

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {}
    }
}
Peterson’s Algorithm

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

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;

        while (flag[j] && victim == i) {} // Is this algorithm correct under sequential consistency?
    }
}

Is this algorithm correct under sequential consistency?
What could go wrong?

class TwoThreadLockFlags {
    
    static volatile boolean[] flag = new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        while (flag[j]) {} // wait
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
What could go wrong?

class TwoThreadLockVolatile {

    static volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i; // wait for the other
        while (victim == i) {} 
    }

    public void unlock() {
    }
}
Filter Algorithm

• There are $n-1$ waiting rooms called “levels”
• One thread gets blocked at each level if many threads try to enter
Filter Lock

class FilterLock {
    int[] level;
    volatile int[] victim;

    public FilterLock() {
        level = new int[n];
        victim = new int[n];
        for (int i = 0; i < n; i++) {
            level[i] = 0;
        }
    }
    public void unlock() {
        int me = ThreadID.get();
        level[me] = 0;
    }
}
Filter Lock

```java
public void lock() {
    int me = ThreadID.get();
    for (int i = 1; i < n; i++) {
        level[me] = i; // visit level i
        victim[i] = me; // Thread me is a good guy!
        // spin while conflict exits
        while ((∃k != me) level[k] >= i && victim[i] == me) {
        }
    }
}
```
Fairness

• Starvation freedom is good, but maybe threads shouldn’t wait too much...

• For example, it would be great if we could order threads by the order in which they performed the first step of the `lock()` method
Bounded Waiting

• Divide lock() method into two parts
  • Doorway interval ($D_A$) – finishes in finite steps
  • Waiting interval ($W_A$) – may take unbounded steps

r-Bounded Waiting

For threads A and B: if $D_A^k \rightarrow D_B^j$, then $CS_A^k \rightarrow CS_B^{j+r}$
Lamport’s Bakery Algorithm

class Bakery implements Lock {
    boolean[] flag;
    Label[] label;

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}

public Bakery(int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i<n; i++) {
        flag[i] = false;
        label[i] = 0;
    }
}
Lamport’s Bakery Algorithm

```java
public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while ((∃k != i) flag[k] && (label[k], k) << (label[i], i)) {} 
}
```
Lamport’s Fast Lock

• Programs with highly contended locks are likely to not scale
• **Insight:** Ideally spin locks should be free of contention

• Idea
  • Two lock fields x and y
  • Acquire: Thread t writes its id to x and y and checks for intervening writes
Lamport’s Fast Lock

class LFL implements Lock {
    private int x, y;
    boolean[] trying;

    LFL() {
        y = ⊥;
        for (int i = 0; i<n; i++) {
            trying[i] = false;
        }
    }

    public void unlock() {
        y = ⊥;
        trying[ThreadID.get()] = false;
    }
}
Lamport’s Fast Lock

public void lock() {
    int self = ThreadID.get();
    start:
        trying[self] = true;
    x = self;
    if (y != ⊥) {
        trying[self] = false;
        while (y != ⊥) {} // spin
        goto start;
    } 
    y = self;
}

if (x != self) {
    trying[self] = false;
    for (i ∈ T) {
        while (trying[i] == true) { // spin
        }
    }
    if (y != self) {
        while (y != ⊥) {} // spin
        goto start;
    }
}
Evaluation Lock Performance

• Lock acquisition latency

• Space overhead

• Fairness

• Bus traffic
Atomic Instructions in Hardware
Hardware Locks

- Locks can be completely supported by hardware
  - Not popular on bus-based machines

- Ideas:
  - Have a set of lock lines on the bus, processor wanting the lock asserts the line, others wait, priority circuit used for arbitrating
  - Special lock registers, processors wanting the lock acquired ownership of the registers

- What could be some problems?
Common Atomic (RMW) Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Implementation</th>
</tr>
</thead>
</table>
| test_and_set      | bool TAS(bool* loc):  
|                   |   atomic {  
|                   |     tmp := *loc;  
|                   |     *loc := true;  
|                   |     return tmp;  
|                   | } |
| swap              | word Swap(word* a, word b):  
|                   |   atomic {  
|                   |     tmp := *a;  
|                   |     *a := b;  
|                   |     return tmp;  
|                   | } |
| fetch_and_inc     | int FAI(int* loc):  
|                   |   atomic {  
|                   |     tmp := *loc;  
|                   |     *loc := tmp+1;  
|                   |     return tmp;  
|                   | } |
| fetch_and_add     | int FAA(int* loc, int n):  
|                   |   atomic {  
|                   |     tmp := *loc;  
|                   |     *loc := tmp+n;  
|                   |     return tmp;  
|                   | } |
Common Atomic (RMW) Instructions

```c
bool CAS(word* loc, world old, word new):
    atomic {
        res := (*loc == old);
        if (res)
            *loc := new;
        return res;
    }
```

Common Atomic (RMW) Instructions

compare_and_swap \([x86, \text{IA-64, SPARC}]\)

\[
\text{bool CAS(word* loc, world old, word new):} \\
\begin{align*}
\text{atomic} & \{ \\
\text{res} & := (*\text{loc} == \text{old}); \\
\text{if} & (\text{res}) \\
\text{*loc} & := \text{new}; \\
\text{return} & \text{res}; \\
\} 
\end{align*}
\]

How can you implement fetch_and_func() with CAS?
## Common Atomic (RMW) Instructions

<table>
<thead>
<tr>
<th>load_linked/store_conditional</th>
<th>[POWER, MIPS, ARM]</th>
</tr>
</thead>
</table>

```plaintext
together

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>word LL(word* a):</code></td>
<td>atomic { remember a; return *a; }</td>
</tr>
<tr>
<td><code>bool SC(word* a, word w):</code></td>
<td>atomic { res := (a is remembered, and has not been evicted since LL) if (res) *a = w; return res; }</td>
</tr>
</tbody>
</table>
```

**Notes:**
- `LL` and `SC` are atomic operations.
- `LL` loads a value from memory, ensuring it is consistent with the current state.
- `SC` stores a value into memory, ensuring it is consistent with the current state.

**Power, MIPS, ARM**

- These instructions support these architectures, providing atomicity and order independence for memory accesses.

---

*CS636 Swarnendu Biswas*
Common Atomic (RMW) Instructions

<table>
<thead>
<tr>
<th>load_linked/store Conditional</th>
<th>[POWER, MIPS, ARM]</th>
</tr>
</thead>
</table>

```c
word LL(word* a):
    atomic {
        remember a;
        return *a;
    }

bool SC(word* a, word w):
    atomic {
        res := (a is remembered, and has not been evicted since LL)
        if (res)
            *a = w;
        return res;
    }
```

How can you implement `fetch_and_func()` with LL/SC?
ABA Problem

**void push(node** top, node* new):**

node* old
repeat
old := *top
new->next := old
until CAS(top, old, new)

**node* pop(node** top):**

node* old, new
repeat
old := *top
if old = null return null
new := old->next
until CAS(top, old, new)
return old
ABA Problem

void push(node** top, node* new):
    node* old
    repeat
        old := *top
        new->next := old
    until CAS(top, old, new)

node* pop(node** top):
    node* old, new
    repeat
        old := *top
        if old = null return null
        new := old->next
    until CAS(top, old, new)
    return old
ABA Problem

void push(node** top, node* new):
    node* old;
    do {
        old = *top;
        new->next = old;
    } while (!CAS(top, old, new));

node* pop(node** top):
    node* old, new;
    do {
        old = *top;
        if (old == null) return null;
        new = old->next;
    } while (!CAS(top, old, new));
    return old;
Common Atomic (RMW) Instructions

**compare_and_swap**
- Cannot detect ABA

**load_linked/store_conditional**
- Guaranteed to fail
- SC can experience spurious failures
  - E.g., Cache miss, branch misprediction
## Common Atomic (RMW) Instructions

<table>
<thead>
<tr>
<th>load_linked/store_conditional</th>
<th>[POWER, MIPS, ARM]</th>
</tr>
</thead>
</table>

**word LL(word* a):**

```c
atomic {
    remember a;
    return *a;
}
```

**bool SC(word* a, word w):**

```c
atomic {
    res := (a is remembered, and has not been evicted since LL)
    if (res)
        *a = w;
    return res;
}
```

How can you reduce spurious failures in your `fetch_and_func()` implementation with LL/SC?
Centralized Mutual Exclusion Algorithms
Test-And-Set

• Atomically tests and sets a word
  • For example, swaps one for zero and returns the old value

• java.util.concurrent.AtomicBoolean::getAndSet(bool val)

• Bus traffic?
• Fairness?

```java
bool TAS(bool* loc) {
    bool res;
    atomic {
        res = *loc;
        *loc = true;
    }
    return res;
}
```
Spin Lock with TAS

class SpinLock {
    bool loc = false;

    public void lock() {
        while (TAS(&loc)) {
            // spin
        }
    }

    public void unlock() {
        loc = false;
    }
}
Test-And-Test-And-Set

• Keep reading the memory location till the location **appears** unlocked
• Reduces bus traffic – why?

```c
int TATAS_GET(int loc);
int TAS(int loc);

do {
    while (TATAS_GET(loc)) {
    
    }
} while (TAS(loc));
```
Exponential Backoff

Larger number of unsuccessful retries
→ Higher the contention
→ Longer backoff
  • Possibly double each time till a given maximum
Spin Lock with TAS and Backoff

class SpinLock {
    bool loc = false;
    const in MIN = ...;
    cost int MUL = ...;
    const int MAX = ...;

    public void unlock() {
        loc = false;
    }
}

public void lock() {
    int backoff = MIN;
    while (TAS(&loc)) {
        pause(backoff);
        backoff = min(backoff * MUL, MAX);
    }
}
Challenges with Exponential Backoff

Larger number of unsuccessful retries
→ Higher the contention
→ Longer backoff

What can be some problems with this?
Fairness with TAS and TATAS Locks
Ticket Lock

• Grants access to threads based on FCFS
• Uses fetch_and_inc()
class TicketLock implements Lock {

    int next_ticket = 0;
    int now_serving = 0;

    public void unlock() {
        now_serving++;
    }

    public void lock() {
        int my_ticket = FAI(&next_ticket);
        while (now_serving != my_ticket) {}
Ticket Lock

class TicketLock implements Lock {
    int next_ticket = 0;
    int now_serving = 0;

    public void unlock() {
        now_serving++;
    }

    public void lock() {
        int my_ticket = FAI(&next_ticket);
        while (now_serving != my_ticket) {}}

    }
Scalable Spin Locks
Queued Locks

• Key idea

  • Instead of contending on a single “now_serving” variable, make threads wait in a queue (i.e., FCFS).
  • Each thread knows its order in the queue.

Implementations

  • Implement a queue using arrays
    • Statically or dynamically allocated depending on the number of threads
  • Each thread spins on its own lock (i.e., array element), and knows the successor information
public class ArrayLock implements Lock {
    AtomicInteger tail;
    boolean[] flag;
    ThreadLocal<Integer> mySlot = …;

    public ArrayLock(int size) {
        tail = new AtomicInteger(0);
        flag = new boolean[size];
        flag[0] = true;
    }

    public void lock() {
        int slot = FAI(tail);
        mySlot.set(slot);
        while (!flag[slot]) {} 
    }

    public void unlock() {
        int slot = mySlot.get();
        flag[slot] = false;
        flag[slot+1] = true;
    }
}
Queued Locks

• Key idea
  • Instead of contending on a single “now_serving” variable, make threads wait in a queue.
  • Each thread knows its order in the queue.

Implementations
  • Implement a queue using arrays
    • Statically or dynamically allocated depending on the number of threads
    • Each thread spins on its own lock (i.e., array element), and knows the successor information

What could be a few disadvantages of array-based Queue locks?
public class ArrayLock implements Lock {
    AtomicInteger tail;
    boolean[] flag;
    ThreadLocal<Integer> mySlot = …;

    public ArrayLock(int size) {
        tail = new AtomicInteger(0);
        flag = new boolean[size];
        flag[0] = true;
    }

    public void lock() {
        int slot = FAI(tail);
        mySlot.set(slot);
        while (!flag[slot]) {} }

    public void unlock() {
        int slot = mySlot.get();
        flag[slot] = false;
        flag[slot+1] = true;
    }
}
MCS Queue Lock

- Proposed by Mellor-Crumney and Scott [1991]
- Uses linked lists instead of arrays
- Space required to support $n$ threads and $k$ locks: $O(n+k)$
- State-of-art scalable FIFO locks
MCS Queue Lock

class QNode {
    QNode next;
    bool waiting;
}

public class MCSLock implements Lock {
    Node tail = null;
    ThreadLocal<QNode> myNode = …;

    public void lock() {
        QNode node = myNode.get();
        QNode prev = swap(tail, node);
        if (prev != null)
            node.waiting = true;
        prev.next = node;
        while (node.waiting) {}
    }

    public void unlock() {
        QNode node = myNode.get();
        QNode succ = node.next;
        if (succ == null)
            if (CAS(tail, node, null))
                return;
        do {
            succ = node.next;
        } while (succ == null);
        succ.waiting = false;
    }
}
MCS Lock Operations

Lock

tail
MCS Lock Operations

- Lock
- Owns the critical section
MCS Lock Operations

Lock

Lock

A

B

tail

tail
MCS Lock Operations

tail

Lock → A → B → C
MCS Lock Operations

A → B → C

Lock → A

B → C

Lock → B

C
Which Spin Lock should I use?

• Limited use of load-store-only locks
• Limited contention (e.g., few threads)
  • TAS spin locks with exponential backoff
  • Ticket locks
• High contention
  • MCS lock
Miscellaneous Lock Optimizations
Reentrant Locks

- A lock that can be **re-acquired** by the owner thread
- Freed after an equal number of releases

```java
public class ParentWidget {
    public synchronized void doWork() {
        ...
    }
}
```

```java
public class ChildWidget extends ParentWidget {
    public synchronized void doWork() {
        ...
        super.doWork();
        ...
    }
}
```
Lazy Initialization In Single-Threaded Context

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
    ...
}

Lazy Initialization In Multithreaded Context

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
    ...
}

Can we optimize the initialization pattern?

1. Check if helper is initialized. If yes, return.
2. If no, then obtain a lock.
3. Double check whether the helper has been initialized. If yes, return.
   • Perhaps concurrently initialized in between Steps 1 and 2.
4. Initialize helper, and return.
Broken Usage of Double Checked Locking

class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized (this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
    ...
}
One Correct Use of Double Checked Locking

class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized (this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
    ...
}
Reader-Writer Locks

- Many objects are read concurrently
  - Updated only a few times

- Reader lock
  - No thread holds the write lock

- Writer lock
  - No thread holds the reader or writer locks

```java
public interface RWLock {
    public void readerLock();
    public void readerUnlock();

    public void writerLock();
    public void writerUnlock();
}
```
## Issues to Consider in Reader-Writer Locks

<table>
<thead>
<tr>
<th>Design choices</th>
<th>Release preference order</th>
<th>Writer releases lock, both readers and writers are queued up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incoming readers</td>
<td>Writers waiting, and new readers are arriving</td>
</tr>
<tr>
<td></td>
<td>Downgrading</td>
<td>Can a thread acquire a read lock without releasing the write lock?</td>
</tr>
<tr>
<td></td>
<td>Upgrading</td>
<td>Can a read lock be <strong>upgraded</strong> to a write lock?</td>
</tr>
</tbody>
</table>
Reader-Writer Locks

- Reader or writer preference
  - Allows starvation of non-preferred threads

readerLock():
    acquire(rd)
    rdrs++
    if rdrs == 1:
      acquire(wr)
    release(rd)

readerUnlock():
    acquire(rd)
    rdrs--
    if rdrs == 0:
      release(wr)
    release(rd)

writerLock():
    acquire(wr)

writerUnlock():
    release(wr)
Reader-Writer Lock With Reader-Preference

class RWLock {
    int n = 0;
    const int WR_MASK = 1;
    const int RD_INC = 2;

    public void writerLock() {
        while (!CAS(&n, 0, WR_MASK)) {
        }
    }

    public void writerUnlock() {
        FAA(&n, -WR_MASK);
    }

    public void readerLock() {
        FAA(&n, RD_INC);
        while ((n & WR_MASK) == 1) {
        }
    }

    public void readerUnlock() {
        FAA(&n, -RD_INC);
    }
}
Asymmetric Locks

• Often objects are locked by at most one thread

• Biased locks
  • JVMs use biased locks, the acquire/release operations on the owner threads are cheaper
    • Usually biased to the first owner thread
  • Synchronize only when the lock is contended, need to take care of several subtle issues
  • `-XX:+UseBiasedLocking` in HotSpot JVM

Monitors
Using Locks to Access a Bounded Queue

• Suppose I have a **bounded** FIFO queue

• Many producer threads and one consumer thread access the queue

```java
mutex.lock();
try {
    queue.enq(x);
} finally {
    mutex.unlock();
}
```
Using Locks to Access a Bounded Queue

• Suppose I have a **bounded** FIFO queue
• Many producer threads and one consumer thread access the queue

```java
mutex.lock();
try {
    queue.enq(x);
} finally {
    mutex.unlock();
}
```

What could be some problems?
Monitors to the Rescue!

• Combination of methods, mutual exclusion locks and condition variables

• Provides **mutual exclusion for methods**

• Provides the possibility to **wait for a condition (cooperation)**

```java
public synchronized void enqueue()
{
    queue.enq(x);
}
```
Condition Variables in Monitors

• Have an associated queue

• Operations
  • wait
  • notify (signal)
  • notifyAll (broadcast)
Condition Variable Operations

wait var, mutex

- Make the thread wait until a condition $COND$ is true
- Releases the **monitor’s mutex**
- Moves the thread to var’s wait queue
- Puts the thread to sleep
- **Steps 1-3 are atomic to prevent race conditions**
- When the thread wakes up, it is assumed to hold mutex
Condition Variable Operations

**notify var**
- Invoked by a thread to assert that $COND$ is true
- Moves one or more threads from the wait queue to the ready queue

**notifyAll var**
- Moves all threads from wait queue to the ready queue
## Signaling Policies

<table>
<thead>
<tr>
<th><strong>Signal and continue (SC)</strong></th>
<th>Signaler thread holds the lock. Java implements SC only.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signal and wait (SW)</strong></td>
<td>Signaler thread needs to reacquire the lock, signaled thread can continue execution</td>
</tr>
<tr>
<td><strong>Signal and urgent wait (SU)</strong></td>
<td>Like SW, but signaler thread gets to go after the signaled thread</td>
</tr>
<tr>
<td><strong>Signal and exit (SX)</strong></td>
<td>Signaler exits, signaled thread can continue execution.</td>
</tr>
</tbody>
</table>
Using Monitors

• Have an associated queue
• Operations
  • wait
  • notify (signal)
  • notifyAll (broadcast)

```
acquire(mutex)
while (!COND) {
  wait(var, mutex)
}
...
/* CRITICAL SECTION */
...
notify(var)/notifyAll(var)
release(mutex)
```
Producer-Consumer with Monitors

Queue q;
Mutex mtx; // Has associated queue
CondVar empty, full;

producer:
  while true:
    data = new Data(...);
    acquire(mtx);
    while q.isFull():
      wait(full, mtx);
    q.enq(data);
    notify(empty);
    release(mtx);

consumer:
  while true:
    acquire(mtx)
    while q.isEmpty():
      wait(empty, mtx);
    data = q.deq();
    notify(full);
    release(mtx);
    ...
    ...

CS636
Swarndu Biswas
85
Contrast with Producer-Consumer with Spin Locks

Queue q;
Mutex mtx;

producer:
while true:
    data = new Data(...);
    acquire(mtx);
    while q.isFull():
        release(mtx);
    ...
    acquire(mtx);
    q.enq(data);
    release(mtx);

consumer:
while true:
    acquire(mtx);
    while q.isEmpty():
        release(mtx);
    ...
    acquire(mtx);
    data = q.deq();
    release(mtx);
    ...
    ...

CS636
Swarndu Biswas
86
Semaphore Implementation with Monitors

```c
int numRes = N;
Mutex mtx;
CondVar zero;

P:
    acquire(mtx);
    while numRes == 0:
        wait(zero, mtx);
    assert numRes > 0
    numRes--;
    release(mtx);

V:
    acquire(mtx);
    numRes++;
    notify(zero);
    release(mtx);
```
Reader-Writer Locks with Reader-Preference

- Reader or writer preference
  - Allows starvation of non-preferred threads

readerLock():
  acquire(rd)
  rdrs++
  if rdrs == 1:
    acquire(wr)
  release(rd)

readerUnlock():
  acquire(rd)
  rdrs--
  if rdrs == 0:
    release(wr)
  release(rd)

writerLock():
  acquire(wr)

writerUnlock():
  release(wr)
Reader-Writer Locks

• Reader or writer preference
  • Allows starvation of non-preferred threads

```
readerLock():
  acquire(rd)
  rdrs++
  if rdrs == 1:
    acquire(wr)
  release(rd)

readerUnlock():
  acquire(rd)

writerLock():
  acquire(wr)

writerUnlock():
  release(wr)
```

How can we construct a Reader-Writer lock with writer-preference?
Monitors in Java

- Java provides built-in support for monitors
  - `synchronized` blocks and methods
  - `wait()`, `notify()`, and `notifyAll()`
- Each object can be used as a monitor

Bounded Buffer with Monitors in Java

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BoundedBuffer {
    private final String[] buffer;
    private final int capacity; // Constant, length of buffer
    private int count; // Current size
    private final Lock lock = new ReentrantLock();
    private final Condition full = new Condition();
    private final Condition empty = new Condition();
}
```
Bounded Buffer with Monitors in Java

public void addToBuffer() ... {
    lock.lock();
    try {
        while (count == capacity)
            full.await();

        ... 
        ... 
        empty.signal();
    } finally {
        lock.unlock();
    }
}

public void removeFromBuffer() ... {
    lock.lock();
    try {
        while (count == 0)
            empty.await();

        ... 
        ... 
        full.signal();
    } finally {
        lock.unlock();
    }
}
References