Outline

• Semaphores
• Monitors
  • Mesa vs. Hoare
• Communicating sequential process
• Use cases
  • Bounded buffer
  • Reader writer lock
Recap: Locks Using Interrupts

If one thread is in critical section, no other activity (including OS) can run!

lock.Acquire();
...
critical section;
...
lock.Release();

Acquire() {
    disable interrupts;
}

int value = 0;
Acquire() {
    // Short busy-wait time
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go_to_sleep(); //??
    } else {
        value = 1;
    }
    enable interrupts;
}

Release() {
    // Short busy-wait time
    disable interrupts;
    if (threads on wait queue) {
        take thread off wait-queue
        place it on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}
Recap: Locks Using test&set

Threads waiting to enter critical section busy-wait

```
int guard = 0;
int value = 0;
Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == 1) {
        put thread on wait-queue;
        go_to_sleep() & guard = 0;
    } else {
        value = 1;
        guard = 0;
    }
}

Release() {
    // Short busy-wait time
    while (test&set(guard));
    if (threads on wait queue) {
        take thread off wait-queue
        place it on ready queue;
    } else {
        value = 0;
    }
    guard = 0;
}
```

```
lock.Acquire();
...
critical section;
...
lock.Release();
```

```
int value = 0
Acquire() {
    while (test&set(value));
}

Release() {
    value = 0;
}
```
Rules for Using Locks

• Lock should be initially free
• Always acquire before accessing shared data
  • Best place for acquiring lock: beginning of procedure!
• Always release lock after finishing with shared data
  • Best place for releasing lock: end of procedure!
  • Only the lock holder can release
  • DO NOT throw lock for someone else to release
• Never access shared data without lock
  • Danger! Don’t do it even if it’s tempting!
Acquire Lock Before Accessing Shared Data, ALWAYS!

```c
getP() {
    if (p == NULL) {
        lock.acquire();
        if (p == NULL) {
            tmp = malloc(sizeof(...));
            tmp->field1 = ...
            tmp->field2 = ...
            p = tmp;
        }
        lock.release();
    }
    return p;
}
```

- Safe but expensive solution is

```c
getP() {
    lock.acquire();
    if (p == NULL) {
        tmp = malloc(sizeof(...));
        tmp->field1 = ...
        tmp->field2 = ...
        p = tmp;
    }
    lock.release();
    return p;
}
```

- Does this work?
  - No! Compiler/HW could make `p` point to `tmp` before its fields are set
  - This is called *double-checked locking*
Producer-Consumer with Bounded Buffer

• Problem definition
  • Producer puts things into shared buffer
  • Consumer takes them out
  • Need to synchronize access to this buffer
  • Producer needs to wait if buffer is full
  • Consumer needs to wait if buffer is empty

• Example 1: GCC compiler
  • cpp | cc1 | cc2 | as | ld

• Example 2: newspaper vending machine
  • Producer can put limited number of newspapers in machine
  • Consumer can’t take newspaper out if machine is empty
Bounded Buffer
Correctness Constraints

- Consumer must wait for producer if buffer is empty (scheduling constraint)
- Producer must wait for consumer if buffer is full (scheduling constraint)
- Only one thread can manipulate buffer at any time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine delivery person filling the vending machine, and somebody comes and tries to stick their money into the machine
Bounded Buffer Implementation with Locks

try_to_produce(item) {
    lock.acquire();
    success = FALSE;
    if (queue.size() < MAX) {
        queue.enqueue(item);
        success = TRUE;
    }
    lock.release();
    return success;
}

try_to_consume() {
    lock.acquire();
    item = NULL;
    if (!queue.isEmpty())
        item = queue.dequeue();
    lock.release();
    return item;
}

• If try_to_consume() returns NULL, do we know buffer is empty?
  • No! Producer might have filled buffer
  • We only know buffer was empty when we tried
  • After releasing lock our knowledge of buffer might not be accurate – we only know state of buffer while holding the lock!

• Is it a good idea to do while(try_to_consume() != NULL)?
  • This will delay producer’s thread from putting items on buffer – bad for everyone!
Where Are We Going with Synchronization?

<table>
<thead>
<tr>
<th>Programs</th>
<th>Shared Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level API</td>
<td>Locks  <strong>Semaphores</strong>  Monitors  Send/Receive</td>
</tr>
<tr>
<td>Hardware</td>
<td>Load/Store  Disable Interrupts  Test&amp;Set  Compare&amp;Swap</td>
</tr>
</tbody>
</table>

- We will see how we can implement various higher-level synchronization primitives using atomic operations
  - Everything is quite painful if load/store are the only atomic primitives
  - Hardware needs to provide more primitives useful at user-level
Semaphores

• Semaphores are a kind of generalized lock
  • First defined by Dijkstra in late 60s
  • Main synchronization primitive used in original UNIX

• Semaphore has non-negative integer value and 2 operations
  • $P()$: atomic operation that waits for semaphore to become positive, then decrements it by one
  • $V()$: atomic operation that increments semaphore by one, waking up a waiting $P()$, if any
  • In Dutch, $P$ stands for proberen (to test) and $V$ stands for verhogen (to increment)
Semaphores (cont.)

• Semaphores are like integers, except
  • No negative values
  • Only available operations are P & V
    (cannot read/write value, except to set it initially)
  • Operations must be atomic
    • Two P’s together can’t decrement value below zero
    • Thread going to sleep in P won’t miss wakeup from V
      (even if they both happen at same time)

• Example: semaphore from railway analogy (initialized to 2)
Example Uses of Semaphores

• Mutual exclusion with binary semaphore (initialized to 1)

```c
semaphore.P();
// Critical section goes here
semaphore.V();
```

• Scheduling constraints (initialized to 0)

```c
thread_join(...) {
    semaphore.P();
}
thread_exit() {
    semaphore.V();
}
```
Recall: Bounded Buffer Correctness Constraints

- Consumer must wait for producer if buffer is empty (scheduling constraint)
- Producer must wait for consumer if buffer is full (scheduling constraint)
- Only one thread can manipulate buffer at any time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine delivery person filling the vending machine, and somebody comes and tries to stick their money into the machine
Bounded Buffer Implementation with Semaphore

General rule of thumb is to use separate semaphore for each constraint

Semaphore emptySlots = MAX;
Semaphore fullSlots = 0;
Semaphore mutex = 1;

produce(item) {
    emptySlots.P();
    mutex.P();
    queue.enqueue(item);
    mutex.V();
    fullSlots.V();
}

consume() {
    fullSlots.P();
    mutex.P();
    item = queue.dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
Discussion about Solution

- Why asymmetry?
- Producer does: `emptySlots.P()`, `fullSlots.V()`
- Consumer does: `fullSlots.P()`, `emptySlots.V()`

Decrease # of empty slots
Increase # of occupied slots
Decrease # of occupied slots
Increase # of empty slots
Discussion about Solution (cont.)

- Is order of P’s important?
  - Yes! Can cause deadlock

- Is order of V’s important?
  - No, it only might affect scheduling efficiency

produce(item) {
    mutex.P();
    emptySlots.P();
    queue.enqueue(item);
    mutex.V();
    fullSlots.V();
}

consume() {
    fullSlots.P();
    mutex.P();
    item = queue.dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
“During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.”

Dijkstra “The structure of the ’THE’-Multiprogramming System” Communications of the ACM v. 11 n. 5 May 1968.”
Motivation for Monitors and Condition Variables

- **Problem**: semaphores are dual purpose:
  - They are used for both mutex and scheduling constraints
  - Example: the fact that flipping of P’s in bounded buffer gives deadlock is not immediately obvious

- **Solution**: use locks for mutual exclusion and Condition Variables (CV) for scheduling constraints

- **Definition**: monitor is one lock with zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables
Monitor with Condition Variables

• Lock provides mutual exclusion to shared data
  • Always acquire before accessing shared data structure
  • Always release after finishing with shared data
  • Lock initially free

• CV is queue of threads waiting for event inside critical section
  • Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  • Contrast to semaphores: can’t wait inside critical section
Condition Variables Operations

- **wait(Lock *lock)**
  - Atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- **signal()**
  - Wake up a waiter, if any
- **broadcast()**
  - Wake up all waiters, if any
Properties of Condition Variables

- Condition variables are memoryless
  - No internal memory except a queue of waiting threads
  - No effect in calling `signal/broadcast` on empty queue
- **ALWAYS** hold lock when calling `wait()`, `signal()`, `broadcast()`
  - In Birrell paper, he says you can call `signal()` outside of lock – IGNORE HIM (this is only an optimization)
- Calling `wait()` atomically adds thread to wait queue and releases lock
- Re-enabled waiting threads may not run immediately
  - No atomicity between `signal/broadcast` and the return from `wait`
method_that_waits() {
    lock.acquire();
    // Read/write shared state
    while (!testSharedState())
        cv.wait(&lock);
    // Read/write shared state
    lock.release();
}

method_that_signals() {
    lock.acquire();
    // Read/write shared state
    // If testSharedState is now true
    cv.signal();
    // Read/write shared state
    lock.release();
}
Bounded Buffer Implementation with Monitors

Lock lock;
CV emptyCV, fullCV;

produce(item) {
    lock.acquire(); // get lock
    while (queue.size() == MAX) // wait until there is space
        fullCV.wait(&lock);
    queue.enqueue(item);
    emptyCV.signal(); // signal waiting costumer
    lock.release(); // release lock
}

text

consume() {
    lock.acquire(); // get lock
    while (queue.isEmpty()) // wait until there is item
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal(); // signal waiting producer
    lock.release(); // release lock
    return item;
}
Question

• Does $k^{th}$ call to `consume()` return item put by $k^{th}$ call to `produce()`?
  • No! thread calling `wait()` must re-acquire lock after waking up; before woken-up thread re-acquire lock, a newly arrived thread could acquire lock and consume item
Mesa vs. Hoare Monitors

- Consider piece of `consume()` code

```java
while (queue.isEmpty())
    emptyCV.wait(&lock);
```

- Why didn't we do this?

```java
if (queue.isEmpty())
    emptyCV.wait(&lock);
```

- **Answer**: it depends on the type of scheduling
  - Hoare-style
  - Mesa-style
Hoare Monitors

- Signaler gives up lock and processor to waiter – waiter runs immediately
- Waiter gives up lock and processor back to signaler when it exits critical section or if it waits again

```java
... lock.acquire()
... 
... dataready.signal();
... lock.release();

lock.acquire()
... if (queue.isEmpty())
... emptyCV.wait(&lock);
... 
lock.release();
```
Mesa Monitors

- Signaler keeps lock and processor
- Waiter placed on ready queue with no special priority
- Practically, need to check condition again after wait
- Most real operating systems

```java
... lock.acquire()
... dataready.signal();
... lock.release();
```

```
lock.acquire()
... while (queue.isEmpty())
    emptyCV.wait(&lock);
... lock.release();
```
Mesa Monitor: Why “while()”?  

- What if we use “if” instead of “while” in bounded buffer example?

```java
consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}

produce(item) {
    lock.acquire();
    if (queue.size() == MAX)
        fullCV.wait(&lock);
    queue.enqueue(item);
    emptyCV.signal();
    lock.release();
}
```

Use “if” instead of “while”
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

Monitor

lock: FREE
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL ...

T1 (Running)

```java
consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
```
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

Monitor

lock: BUSY (T1)
emptyCV queue \rightarrow\text{NULL}

CPU State

Running: T1
ready queue \rightarrow\text{NULL}
...

T1 (Running)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

```
queue
```

Monitor

`lock: FREE`

`emptyCV queue → T1`

CPU State

`Running:
ready queue → NULL`

```
consume()
{
  lock.acquire();
  if (queue.isEmpty())
    emptyCV.wait(&lock);
  item = queue.dequeue();
  fullCV.signal();
  lock.release();
  return item;
}
```

T1 (Waiting)

`wait(&lock)` puts thread on emptyCV queue and releases lock
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

Monitor

lock: FREE
emptyCV queue → T1

CPU State

Running: T2
ready queue → NULL
...

T1 (Waiting)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}

T2 (Running)

produce(item) {
    lock.acquire();
    if (queue.size()==MAX)
        fullCV.wait(&lock);
    queue.enqueue(item);
    emptyCV.signal();
    lock.release();
}
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

Monitor

CPU State

queue

lock: BUSY (T2)
emptyCV queue → T1

Running: T2
ready queue → NULL ...

T1 (Waiting)

consume() {
lock.acquire();
if (queue.isEmpty())
emptyCV.wait(&lock);
item = queue.dequeue();
fullCV.signal();
lock.release();
return item;
}

T2 (Running)

produce(item) {
lock.acquire();
if (queue.size()==MAX)
fullCV.wait(&lock);
queue.enqueue(item);
emptyCV.signal();
lock.release();
}
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

Monitor

lock: BUSY (T2)
emptyCV queue → NULL

CPU State

Running: T2
ready queue → T1
...

T1 (Ready)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}

T2 (Running)

produce(item) {
    lock.acquire();
    if (queue.size()==MAX)
        fullCV.wait(&lock);
    queue.enqueue(item);
    emptyCV.signal();
    lock.release();
}

signal() wakes up and moves it to ready queue
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

Monitor

lock: BUSY (T2)
emptyCV queue → NULL

CPU State

Running: T2
ready queue → T1,T3 ...

T1 (Ready)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}

T2 (Running)

produce(item) {
    lock.acquire();
    if (queue.size()==MAX)
        fullCV.wait(&lock);
    queue.enqueue(item);
    emptyCV.signal();
    lock.release();
}

T3 (Ready)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

Monitor

CPU State

queue

lock: FREE
emptyCV queue → NULL

Running:
ready queue → T1, T3 ...

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
return item;
}

produce(item) {
    lock.acquire();
    if (queue.size() == MAX)
        fullCV.wait(&lock);
    queue.enqueue(item);
    emptyCV.signal();
    lock.release();
}

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
return item;
}
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

lock: FREE
emptyCV queue → NULL

Monitor

queue → NULL

CPU State

Running: T3
ready queue → T1

T1 (Ready)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}

T3 (Running)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}

T3 is scheduled first
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

- queue

Monitor

- lock: BUSY (T3)
- emptyCV queue → NULL

CPU State

- Running: T3
- ready queue → T1
...
**Mesa Monitor: Why “while()”? (cont.)**

<table>
<thead>
<tr>
<th>App. Shared State</th>
<th>Monitor</th>
<th>CPU State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>queue</strong></td>
<td><strong>lock:</strong> BUSY (T3) \n<strong>emptyCV queue → NULL</strong></td>
<td>Running: T3 \nready queue → T1 ...</td>
</tr>
</tbody>
</table>

**T1 (Ready)**

```java
consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
```

**T3 (Running)**

```java
consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
```
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

```
queue
```

Monitor

```
lock: FREE
emptyCV queue → NULL
```

CPU State

```
Running:
ready queue → T1
...
```

T1 (Ready)

```
consume()
{
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
```

T3 (Terminated)

```
consume()
{
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
```
consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

Monitor

lock: BUSY (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
...

T1 (Running)

consume() {
    lock.acquire();
    if (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}

Error!
## Mesa Monitor: Why “while( )”? (cont.)

### App. Shared State
- queue

### Monitor
- lock: BUSY (T1)
- emptyCV queue → NULL

### CPU State
- Running: T1
- ready queue → NULL ...

---

**T1 (Running)**

```java
consume() {
    lock.acquire();
    while (queue.isEmpty())
        emptyCV.wait(&lock);
    item = queue.dequeue();
    fullCV.signal();
    lock.release();
    return item;
}
```

Check again if empty!
Mesa Monitor: Why “while()”? (cont.)

App. Shared State

```
queue
```

Monitor

```
lock: FREE
emptyCV queue → T1
```

CPU State

```
Running:
ready queue → NULL
...```

T1 (Waiting)

```
consume() {
lock.acquire();
while (queue.isEmpty())
  emptyCV.wait(&lock);
item = queue.dequeue();
fullCV.signal();
lock.release();
return item;
}
```
When waiting upon a Condition, a spurious wakeup is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.

From Java User Manual
Condition Variable vs. Semaphore

- CV's `signal()` has no memory
  - If `signal()` is called before `wait()`, then signal is waisted

- Semaphore's `V()` has memory
  - If `V()` is called before `P()`, `P()` will not wait

- Generally, it's better to use monitors but not always

- Example: interrupt handlers
  - Shared memory is read/written concurrently by HW and kernel
  - HW cannot use SW locks
  - Kernel thread checks for data and calls `wait()` if there is no data
  - HW write to shared memory, starts interrupt handler to then call `signal()`
    - This is called `naked notify` because interrupt handler doesn't hold lock (why?)
  - This may not work if signal comes before kernel thread calls `wait`
  - Common solution is to use semaphores instead
Implementing Condition Variable using Semaphores (Take 1)

```c
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal() {
    semaphore.V();
}
```

• Does this work?
  • No! `signal()` should not have memory!
Implementing Condition Variable using Semaphores (Take 2)

wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal() {
    if (semaphore’s queue is not empty)
        semaphore.V();
}

• Does this work?
  • No! For one, not legal to look at contents of semaphore’s queue.
  • But also, releasing lock and going to sleep should happen atomically –
    signaler can slip in after lock is released, and before waiter is put on
    wait queue, which means waiter never wakes up!
Implementing Condition Variable using Semaphores (Take 3)

Key idea: have separate semaphore for each waiting thread and put semaphores in ordered queue

wait(lock) {
    semaphore = new Semaphore; // a semaphore per waiting thread
    queue.enqueue(semaphore); // queue for waiting threads
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal() {
    if (!queue.isEmpty()) {
        semaphore = queue.dequeue()
        semaphore.V();
    }
}
Readers/Writers Lock

- Motivation: consider shared database with two classes of users
  - Readers: never modify database
  - Writers: read and modify database
- Database can have many readers at the same time
- But there can be only one writer active at a time
Properties of Readers/Writers Lock

• Common variant of mutual exclusion
  • One writer at a time, if no readers
  • Many readers, if no writer

- Correctness constraints
  • Readers can read when no writers
  • Writers can read/write when no readers or writers
  • Only one thread manipulates state of the lock at a time
class ReadWriteLock {
    private:
    Lock lock; // needed to change state vars
    CV okToRead // CV for readers
    CV okToWrite; // CV for writers
    int AW = 0; // # of active writers
    int AR = 0; // # of active readers
    int WW = 0; // # of waiting writers
    int WR = 0; // # of waiting readers
    
    public:
    void acquireRL();
    void releaseRL();
    void acquireWL();
    void releaseWL();
}
Readers/Writers Lock Design Pattern

```java
read() {
    lock.acquireRL();
    // Read shared state
    lock.releaseRL();
}

write() {
    lock.acquireWL();
    // Read/write shared state
    lock.releaseWL();
}
```
acquireRL() {  
    lock.acquire();  // Need lock to change state vars  
    while (AW + WW > 0) {  // Is it safe to read?  
        WR++;  // No! add to # of waiting readers  
        okToRead.wait(&lock);  // Wait on condition variable  
        WR--;  // No longer waiting  
    }  
    AR++;  // Now we are active again  
    lock.release();
}

releaseRL() {  
    lock.acquire();  // No longer active  
    AR--;  
    if (AR == 0 && WW > 0)  // If no active reader,  
        okToWrite.signal();  // wake up waiting writer  
    lock.release();
}
acquireWL() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
}

releaseWL() {
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
Simulation of Readers/Writers Lock

• Consider following sequence of arrivals: R1, R2, W1, R3

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}

write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R1 comes – AR = 0, WR = 0, AW = 0, WW = 0

```java
read()
{
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R1 comes – AR = 0, WR = 0, AW = 0, WW = 0

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R1 comes – AR = 0, WR = 0, AW = 0, WW = 0

read() {
    lock.acquire()
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
Simulation of Readers/Writers Lock

• R1 comes – AR = 1, WR = 0, AW = 0, WW = 0

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R1 comes — AR = 1, WR = 0, AW = 0, WW = 0

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- $R_1$ comes — $AR = 1$, $WR = 0$, $AW = 0$, $WW = 0$

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
}
```

// Read

```java
lock.acquire();
AR--;
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.release();
```
Simulation of Readers/Writers Lock

- R2 comes — AR = 1, WR = 0, AW = 0, WW = 0

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R2 comes – \( AR = 1, \ WR = 0, \ AW = 0, \ WW = 0 \)

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

• R2 comes – AR = 1, WR = 0, AW = 0, WW = 0

    read() {
        lock.acquire();
        while (AW + WW > 0) {
            WR++;
            okToRead.wait(&lock);
            WR--;
        }
        AR++;
        lock.release();

        // Read
        lock.acquire();
        AR--;
        if (AR == 0 && WW > 0)
            okToWrite.signal();
        lock.release();
    }
Simulation of Readers/Writers Lock

- R2 comes — $AR = 2$, $WR = 0$, $AW = 0$, $WW = 0$

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R2 comes — \( AR = 2, \ WR = 0, \ AW = 0, \ WW = 0 \)

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++:
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

• R2 comes – AR = 2, WR = 0, AW = 0, WW = 0

read() {
  lock.acquire();
  while (AW + WW > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
  AR++;
  lock.release();
}

lock.acquire();
AR--;
if (AR == 0 && WW > 0)
  okToWrite.signal();
lock.release();
Simulation of Readers/Writers Lock

- W1 comes \( AR = 2, WR = 0, AW = 0, WW = 0 \)

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

- W1 comes – AR = 2, WR = 0, AW = 0, WW = 0

```java
class Write {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
}
```

```java
// Read and Write
lock.acquire();
AW--;
if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.release();
```
Simulation of Readers/Writers Lock

• W1 comes — AR = 2, WR = 0, AW = 0, WW = 0

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
}
// Read and Write
lock.acquire();
AW--;
if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.release();
```
Simulation of Readers/Writers Lock

- \( W1 \) comes — \( AR = 2, WR = 0, AW = 0, WW = 1 \)

```c
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

• W1 comes – AR = 2, WR = 0, AW = 0, WW = 1

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
    // Read and Write
    lock.acquire();
    AW--;  
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```

W1 cannot start because of readers, so it releases lock and goes to sleep
Simulation of Readers/Writers Lock

- R3 comes — AR = 2, WR = 0, AW = 0, WW = 1

```java
def read()
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
```

Simulation of Readers/Writers Lock

• R3 comes – \( AR = 2, \ WR = 0, \ AW = 0, \ WW = 1 \)

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

• R3 comes — AR = 2, WR = 0, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R3 comes – AR = 2, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R3 comes – AR = 2, WR = 1, AW = 0, WW = 1

read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read

    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}

Status:
- R1 and R2 still reading
- W1 and R3 waiting on okToWrite
  and okToRead, respectively
Simulation of Readers/Writers Lock

- R2 is done reading – AR = 2, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

• R2 is done reading – AR = 1, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

• R2 is done reading – AR = 1, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R2 is done reading – AR = 1, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R1 is done reading – AR = 1, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R1 is done reading \( - AR = 0, WR = 1, AW = 0, WW = 1 \)

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R1 is done reading – AR = 0, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--:
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
```
Simulation of Readers/Writers Lock

• R1 is done reading – AR = 0, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```

All readers are finished, R1 signals waiting writer – note, R3 is still waiting
**Simulation of Readers/Writers Lock**

- R1 is done reading – AR = 0, WR = 1, AW = 0, WW = 1

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
}
```

```java
// Read
lock.acquire();
AR--;
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.release();
```
Simulation of Readers/Writers Lock

- W1 gets a signal — AR = 0, WR = 1, AW = 0, WW = 1

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```

W1 gets signal from R1
Simulation of Readers/Writers Lock

- \( W1 \) gets a signal: \( AR = 0, WR = 1, AW = 0, WW = 0 \)

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

- W1 gets a signal – AR = 0, WR = 1, AW = 1, WW = 0

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
}

// Read and Write
lock.acquire();
AW--;

if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.release();
```
Simulation of Readers/Writers Lock

• W1 gets a signal — AR = 0, WR = 1, AW = 1, WW = 0

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
    // Read and Write
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

- W1 gets a signal – AR = 0, WR = 1, AW = 1, WW = 0

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
}
```

// Read and Write

```java
// Read and Write

lock.acquire();
AW--;

if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.release();
```
Simulation of Readers/Writers Lock

- $W1$ is done — $AR = 0$, $WR = 1$, $AW = 1$, $WW = 0$

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

- \( W1 \) is done — \( AR = 0, WR = 1, AW = 0, WW = 0 \)

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

• W1 is done — AR = 0, WR = 1, AW = 0, WW = 0

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

- W1 is done – AR = 0, WR = 1, AW = 0, WW = 0

write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
Simulation of Readers/Writers Lock

- W1 is done – AR = 0, WR = 1, AW = 0, WW = 0

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```

No waiting writer, so only signal R3
Simulation of Readers/Writers Lock

• W1 is done – AR = 0, WR = 1, AW = 0, WW = 0

```java
write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;

    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R3 gets a signal — AR = 0, WR = 1, AW = 0, WW = 0

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```

R3 gets signal from W3
Simulation of Readers/Writers Lock

- R3 gets a signal – AR = 0, WR = 0, AW = 0, WW = 0

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

• R3 gets a signal — AR = 1, WR = 0, AW = 0, WW = 0

```c
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
Simulation of Readers/Writers Lock

- R3 finishes – AR = 0, WR = 0, AW = 0, WW = 0

```java
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}

write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}

What if we remove this line?

It works but it’s inefficient, writer wakes up and goes to sleep again when it’s not save to write
read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
    // Read
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.broadcast();
    lock.release();
}

write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
    // Read and Write
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}

What if we turn signal() to broadcast()?

It works but it's inefficient to wake up all writers only for one to becomes active
Readers/Writers Lock Questions

read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;
    if (AR == 0 & WW > 0)
        okToContinue.signal();
    lock.release();
}

What if we turn okToWrite and okToRead into okContinue?

write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToContinue.signal();
    } else if (WR > 0) {
        okToContinue.broadcast();
    }
    lock.release();
}

Signal could be delivered to wrong thread (reader) and get waisted!
Readers/Writers Lock Questions

read() {
    lock.acquire();
    while (AW + WW > 0) {
        WR++;
        okToContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // Read
    lock.acquire();
    AR--;        
    if (AR == 0 && WW > 0)
        okToContinue.broadcast();
    lock.release();
}

write() {
    lock.acquire();
    while (AW + AR > 0) {
        WW++;
        okToContinue.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();

    // Read and Write
    lock.acquire();
    AW--;        
    if (WW > 0) {
        okToContinue.broadcast();
    } else if (WR > 0) {
        okToContinue.broadcast();
    }
    lock.release();
}

Does changing signal() to broadcast() solve the problem?

Yes, but it’s inefficient to wake up all threads for only one to becomes active.
Readers/Writers Lock Questions

Can readers starve?

Yes: writers take priority
Can writers starve?

Yes: a waiting writer may not be able to proceed if another writer slips in between signal and wakeup.
Readers/Writers Lock Without Writer Starvation (Take 1)

acquireWL() {
    lock.acquire();
    while (AW + AR + WW > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
}

• Does this work?
  • No! If there WW is more than zero, then no waiting writer can successfully proceed
Readers/Writers Lock Without Writer Starvation (Take 2)

Idea: keep track of writers' waiting order, allow writer with longest waiting time to proceed

Does this work?

No, Signal can wake up a wrong writer and get waisted!

Inefficient solution is to change `signal()` to `broadcast()`
Readers/Writers Lock Without Writer Starvation (Take 3)

```
numWriters = 0;
nextToGo = 1;

acquireWL() {
    lock.acquire();
    myPos = numWriters++;
    myCV = new CV();
    queue.enqueue(myCV);
    while (AW + AR > 0 ||
            myPos > nextToGo) {
        WW++;
        myCV.wait(&lock);
        WW--;
    }  
    AW++;
    queue.dequeue();
    lock.release();
}

releaseWL() {
    lock.acquire();
    AW--;
    nextToGo++;
    if (WW > 0) {
        queue.head().signal();
    } else if (WR > 0) {  
        okToRead.broadcast();
    }
    lock.release();
}

releaseRL() {
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)  
        queue.head().signal();
    lock.release();
}
```

Idea for efficient solution: have separate CV for each waiting writer and put CV’s in ordered queue
Communicating Sequential Processes (CSP/Google Go)

- Instead of allowing threads to access shared objects, each object is assigned to single corresponding thread
  - Only one thread is allowed to access object’s data
- Threads communicate with each other solely through message-passing
  - Instead of calling method on shared object, threads send messages to object’s corresponding thread with method name and arguments
- Thread waits in a loop, gets messages, and performs operations
- No race condition!
Bounded Buffer Implementation with CSP

while (msg = getNext()) {
    if (msg == GET) {
        if (!queue.isEmpty()) {
            // get item
            // send reply
            // if pending put, do it
            // and send reply
        } else {
            // queue get operation
        }
    } else if (msg == PUT) {
        if (queue.size() < MAX) {
            // put item
            // send reply
            // if pending get, do it
            // and send reply
        } else {
            // queue put operation
        }
    }
}
Locks/CVs vs. CSP

• Create a lock on shared data
  = create a single thread to operate on data

• Call a method on a shared object
  = send a message/wait for reply

• Wait for a condition
  = queue an operation that can’t be completed just yet

• Signal a condition
  = perform a queued operation, now enabled

Execution of procedure with monitor lock is equivalent to processing message in CSP (monitor is, in effect, single-threaded while it is holding lock)
Summary

• Semaphores are like integers with only two operations
  • $P()$: Wait if zero; decrement when becomes non-zero
  • $V()$: Increment and wake a sleeping task (if exists)
  • Can initialize value to any non-negative value
  • Use separate semaphore for each constraint

• Monitors have one lock plus one or more condition variables
  • Always acquire lock before accessing shared data
  • Use condition variables to wait inside critical section
    • Three Operations: $wait()$, $signal()$, and $broadcast()$

• Communicating sequential processes
  • Communicate only via message-passing
Questions?
Acknowledgment

• Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny