Recall: Implement Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int mylock = FREE;
Acquire(&mylock) - wait until lock is free, then grab
Release(&mylock) - Unlock, waking up anyone waiting
```

```c
Acquire(int *lock)
{
    disable interrupts;
    if (*lock == BUSY)
        put thread on wait queue;
        // Go to sleep() & Enab ints!
        // Ints disabled on wakeup
    else {
        *lock = BUSY;
        enable interrupts;
    }
}
```

```c
Release(int *lock)
{
    disable interrupts;
    if (anyone on wait queue)
        take thread off wait queue;
        Place on ready queue;
    else {
        *lock = FREE;
        enable interrupts;
    }
}
```

- Really only works in kernel - why?

Recall: Two Implementations for T&S Lock

```c
int guard = 0;
int mylock = FREE;

Acquire_Ver1()
{
    // Short busy-wait time
    while (test&set(guard));
    if (mylock == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
        // guard == 0 on wakeup!
    } else {
        mylock = BUSY;
        guard = 0; // Only success
    }
}
```

```c
Acquire_Ver2()
{
    // Short busy-wait time
    while (test&set(guard));
    if (mylock == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
        // guard == 1 on wakeup!
    } else {
        mylock = BUSY;
        guard = 0; // Success or fail!
    }
}
```

- Brief busy-wait only to protect lock implementation
  - Note: sleep has to be sure to reset the guard variable
- Must understand value of guard on exiting sleep
  - Post - sleep guard value must be specified!
  - Only Acquire_Ver2() parallel to “disable interrupts” case!
In-Kernel Lock: Simulation

Value: 1
waiters owner
READY

Thread A

lock.Acquire();
if (value == 1) {
put thread on wait-queue;
go to sleep(); //??
} else {
lock.Release();
value = 1;
}

lock.Release();
enable interrupts;

lock.Acquire();
if anyone on wait-queue {
take thread off wait-queue
Place on ready queue;
} else {
value = 0;
}

lock.Release();
enable interrupts;

Thread B

Acquire() {
disable interrupts;
if (value == 1) {
put thread on wait-queue;
go to sleep(); //??
} else {
lock.Release();
value = 1;
}

lock.Release();
enable interrupts;

Release() {
disable interrupts;
if anyone on wait-queue {
take thread off wait-queue
Place on ready queue;
} else {
value = 0;
}

lock.Release();
enable interrupts;

Value: 1
waiters owner
READY

Thread A

lock.Acquire();
if (value == 1) {
put thread on wait-queue;
go to sleep(); //??
} else {
lock.Release();
value = 1;
}

lock.Release();
enable interrupts;

lock.Acquire();
if anyone on wait-queue {
take thread off wait-queue
Place on ready queue;
} else {
value = 0;
}

lock.Release();
enable interrupts;

Thread B

Acquire() {
disable interrupts;
if (value == 1) {
put thread on wait-queue;
go to sleep(); //??
} else {
lock.Release();
value = 1;
}

lock.Release();
enable interrupts;

Release() {
disable interrupts;
if anyone on wait-queue {
take thread off wait-queue
Place on ready queue;
} else {
value = 0;
}

lock.Release();
enable interrupts;
Review: Semaphores

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation
  - Only time can set integer directly is at initialization time

- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:

Review: Full Solution to Bounded Buffer

Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
  emptySlots.P(); // Wait until space
  mutex.P(); // Wait until machine free
  Enqueue(item);
  mutex.V();
  fullSlots.V(); // Tell consumers there is more coke
}

Consumer() {
  fullSlots.P(); // Check if there’s a coke
  mutex.P(); // Wait until machine free
  item = Dequeue();
  mutex.V();
  emptySlots.V(); // tell producer need more
  return item;
}
Motivation for Monitors and Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores
  - Problem is that 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. How do you prove correctness to someone?

- Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints

- Definition: Monitor: a lock and 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: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free

- Condition Variable: a queue of threads waiting for something inside a 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

Operations:
- Wait(&lock): Atomically release lock and go to sleep. Re-acquire lock later, before returning.
- Signal(): Wake up one waiter, if any
- Broadcast(): Wake up all waiters

Rule: Must hold lock when doing condition variable ops!
- In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;

AddToQueue(item) {
    lock.Acquire(); // Lock shared data
    queue.enqueue(item); // Add item
    lock.Release(); // Release Lock
}

RemoveFromQueue() {
    lock.Acquire(); // Lock shared data
    item = queue.dequeue(); // Get next item or null
    lock.Release(); // Release Lock
    return(item); // Might return null
}
```

- Not very interesting use of “Monitor”
  - It only uses a lock with no condition variables
  - Cannot put consumer to sleep if no work!
Complete Monitor Example (with cond. variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
    lock.Acquire(); // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal any waiters
    lock.Release(); // Release Lock
}
```

```
RemoveFromQueue() {
    lock.Acquire(); // Get Lock
    while (queue.isEmpty()) {
        dataready.wait(&lock); // If nothing, sleep
    }
    item = queue.dequeue(); // Get next item
    lock.Release(); // Release Lock
    return(item);
}
```

Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

```
while (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
} item = queue.dequeue(); // Get next item
```

- Why didn't we do this?

```
if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
} item = queue.dequeue(); // Get next item
```

- Answer: depends on the type of scheduling
  - Hoare-style: Named after British logician Tony Hoare
  - Mesa-style: Named after Xerox-Park Mesa Operating System

Administrivia

- Project 1 Design Document
  - Due tomorrow (Wednesday, 2/20) @ 11:59PM

- Midterm on Thursday 2/28 8pm-10pm
  - Room assignments TBD

- Closed book, no calculators, one double-side letter-sized page of handwritten notes
  - Covers Lectures 1-11 (up through Deadlock), readings, homework 1, and project 1

Hoare monitors

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
- Most textbooks

```
lock.Acquire()
...
if (queue.isEmpty()) {
    dataready.signal();
    lock.Acquire()
    dataready.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();
```

Mesa Monitor: Why “while()”?

- Why do we use “while()” instead of “if()” with Mesa monitors?  
  – Example illustrating what happens if we use “if()”, e.g.,

```java
if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
}
```

- We’ll use the synchronized (infinite) queue example

```java
AddToQueue(item) {
    lock.Acquire();
    queue.enqueue(item);
    dataready.signal();
    lock.Release();
}
```

```java
RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
```

Mesa Monitor: Why “while()”?

- RemoveFromQueue() with “if” replaces “while”

T1 (Running)

```
RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
```
RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}

AddToQueue(item) {
    lock.Acquire();
    queue.enqueue(item);
    dataready.signal();
    lock.Release();
}

T1 (Waiting)
wait(&lock) puts thread on dataready queue and releases lock

T2 (Running)
signal() wakes up T1 and moves it on ready queue
### Mesa Monitor: Why “while()”?  

<table>
<thead>
<tr>
<th>App. Shared State</th>
<th>Monitor</th>
<th>CPU State</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>lock: BUSY (T2)</td>
<td>Running: T2</td>
</tr>
<tr>
<td></td>
<td>dataready</td>
<td>Ready queue → T1, T3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>T1 (Ready)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveFromQueue()</td>
<td>lock.Acquire();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (queue.isEmpty()) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dataready.wait(&amp;lock);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>item = queue.dequeue();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock.Release();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return(item);</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>AddToQueue(item)</td>
<td>lock.Acquire();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>queue.enqueue(item);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dataready.signal();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock.Release();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return(item);</td>
<td></td>
</tr>
</tbody>
</table>

| Monitor | |
|---------| |
| App. Shared State | |
| queue             | lock: FREE | |
|                   | dataready  | NULL |
|                   |          | |
| T2 (Running)      |          | |
| RemoveFromQueue() | lock.Acquire(); | |
|                   | if (queue.isEmpty()) { | |
|                   |     | |
|                   |     | dataready.wait(&lock); |
|                   |     | |
|                   |     | item = queue.dequeue(); |
|                   |     | lock.Release(); |
|                   |     | return(item); |
|                   | } | |

| Monitor | |
|---------| |
| App. Shared State | |
| queue             | lock: BUSY (T3) | |
|                   | dataready  | NULL |
|                   |          | |
| T3 (Running)      |          | |
| RemoveFromQueue() | lock.Acquire(); | |
|                   | if (queue.isEmpty()) { | |
|                   |     | |
|                   |     | dataready.wait(&lock); |
|                   |     | |
|                   |     | item = queue.dequeue(); |
|                   |     | lock.Release(); |
|                   |     | return(item); |

### Mesa Monitor: Why “while()”?  

<table>
<thead>
<tr>
<th>App. Shared State</th>
<th>Monitor</th>
<th>CPU State</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>lock: FREE</td>
<td>Running: T3</td>
</tr>
<tr>
<td></td>
<td>dataready</td>
<td>Ready queue → T1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>T3 (Ready)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveFromQueue()</td>
<td>lock.Acquire();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (queue.isEmpty()) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dataready.wait(&amp;lock);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>item = queue.dequeue();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock.Release();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return(item);</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

| Monitor | |
|---------| |
| App. Shared State | |
| queue             | lock: BUSY (T3) | |
|                   | dataready  | NULL |
|                   |          | |
| T3 (Running)      |          | |
| RemoveFromQueue() | lock.Acquire(); | |
|                   | if (queue.isEmpty()) { | |
|                   |     | |
|                   |     | dataready.wait(&lock); |
|                   |     | |
|                   |     | item = queue.dequeue(); |
|                   |     | lock.Release(); |
|                   |     | return(item); |

### Mesa Monitor: Why “while()”?  

<table>
<thead>
<tr>
<th>App. Shared State</th>
<th>Monitor</th>
<th>CPU State</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>lock: BUSY (T3)</td>
<td>Running: T3</td>
</tr>
<tr>
<td></td>
<td>dataready</td>
<td>Ready queue → T1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>T3 (Running)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveFromQueue()</td>
<td>lock.Acquire();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (queue.isEmpty()) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dataready.wait(&amp;lock);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>item = queue.dequeue();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock.Release();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return(item);</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

| Monitor | |
|---------| |
| App. Shared State | |
| queue             | lock: BUSY (T3) | |
|                   | dataready  | NULL |
|                   |          | |
| T3 (Running)      |          | |
| RemoveFromQueue() | lock.Acquire(); | |
|                   | if (queue.isEmpty()) { | |
|                   |     | |
|                   |     | dataready.wait(&lock); |
|                   |     | |
|                   |     | item = queue.dequeue(); |
|                   |     | lock.Release(); |
|                   |     | return(item); |

### Mesa Monitor: Why “while()”?  

<table>
<thead>
<tr>
<th>App. Shared State</th>
<th>Monitor</th>
<th>CPU State</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>lock: FREE</td>
<td>Running: T3</td>
</tr>
<tr>
<td></td>
<td>dataready</td>
<td>Ready queue → T1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>T3 (Running)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveFromQueue()</td>
<td>lock.Acquire();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (queue.isEmpty()) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dataready.wait(&amp;lock);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>item = queue.dequeue();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock.Release();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return(item);</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

| Monitor | |
|---------| |
| App. Shared State | |
| queue             | lock: BUSY (T3) | |
|                   | dataready  | NULL |
|                   |          | |
| T3 (Running)      |          | |
| RemoveFromQueue() | lock.Acquire(); | |
|                   | if (queue.isEmpty()) { | |
|                   |     | |
|                   |     | dataready.wait(&lock); |
|                   |     | |
|                   |     | item = queue.dequeue(); |
|                   |     | lock.Release(); |
|                   |     | return(item); |
|                   | } | |
RemoveFromQueue() {
    lock.acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.release();
    return(item);
}
Mesa Monitor: Why “while()”?

RemoveFromQueue() {
    lock.Acquire();
    while (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}

T1 (Running)

- Replace “if” with “while”

T1 (Ready)

- Check again: Is queue empty?

Mesa Monitor: Why “while()”?

RemoveFromQueue() {
    lock.Acquire();
    while (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}

T1 (Waiting)

- Yup! Back to Sleep

Readers/Writers Problem

- Motivation: Consider a shared database
  - Two classes of users:
    » Readers – never modify database
    » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    » Like to have many readers at the same time
    » Only one writer at a time
Basic Readers/Writers Solution

• Correctness Constraints:
  – Readers can access database when no writers
  – Writers can access database when no readers or writers
  – Only one thread manipulates state variables at a time
• Basic structure of a solution:
  – **Reader()**
    - Wait until no writers
    - Access database
    - Check out – wake up a waiting writer
  – **Writer()**
    - Wait until no active readers or writers
    - Access database
    - Check out – wake up waiting readers or writer
  – State variables (Protected by a lock called “lock”):
    » int AR: Number of active readers; initially = 0
    » int WR: Number of waiting readers; initially = 0
    » int AW: Number of active writers; initially = 0
    » int WW: Number of waiting writers; initially = 0
    » Condition okToRead = NIL
    » Condition okToWrite = NIL

Code for a Reader

```c
Reader() {
    // First check self into system
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Acquire();
    if (AR == 0 && WW > 0) // No other active readers
        okToWrite.signal(); // Wake up one writer
    lock.Release();
}
```

Code for a Writer

```c
Writer() {
    // First check self into system
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    if (AR == 0 &amp; WW > 0) // No other active readers
        okToWrite.signal(); // Wake up one writer
    lock.Release();
}
```

Simulation of Readers/Writers Solution

• Use an example to simulate the solution

• Consider the following sequence of operators:
  – R1, R2, W1, R3

• Initially: AR = 0, WR = 0, AW = 0, WW = 0
Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

```c
Reader() {
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();
  AccessDBase(ReadOnly);
}
```

```c
lock.Acquire();
AR--; if (AR == 0 && WW > 0)
okToWrite.signal();
lock.Release();
```

2/19/19  Kubiatowicz CS162 © UCB Spring 2018  Lec 9.45

Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader() {
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();
  AccessDBase(ReadOnly);
}
```

```c
lock.Acquire();
AR--; if (AR == 0 && WW > 0)
okToWrite.signal();
lock.Release();
```

2/19/19  Kubiatowicz CS162 © UCB Spring 2018  Lec 9.46

Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader() {
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();
  AccessDBase(ReadOnly);
}
```

```c
lock.Acquire();
AR--; if (AR == 0 && WW > 0)
okToWrite.signal();
lock.Release();
```

2/19/19  Kubiatowicz CS162 © UCB Spring 2018  Lec 9.47

Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader() {
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();
  AccessDBase(ReadOnly);
}
```

```c
lock.Acquire();
AR--; if (AR == 0 && WW > 0)
okToWrite.signal();
lock.Release();
```

2/19/19  Kubiatowicz CS162 © UCB Spring 2018  Lec 9.48
Simulation of Readers/Writers Solution

- R1 accessing dbase (no other threads)
  - AR = 1, WR = 0, AW = 0, WW = 0

Reader()
{
  lock.Acquire();
  while ((AW + WW) > 0) // Is it safe to read?
  {
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();

  AccessDBase(ReadOnly);

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

Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
  - AR = 1, WR = 0, AW = 0, WW = 0

Reader()
{
  lock.Acquire();
  while ((AW + WW) > 0) // Is it safe to read?
  {
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();

  AccessDBase(ReadOnly);

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

Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
  - AR = 2, WR = 0, AW = 0, WW = 0

Reader()
{
  lock.Acquire();
  while ((AW + WW) > 0) // Is it safe to read?
  {
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();

  AccessDBase(ReadOnly);

  lock.Acquire();
  AR--; if (AR == 0 && WW > 0)
  okToWrite.signal();
  lock.Release();
}
Simulation of Readers/Writers Solution

- **R2 comes along (R1 accessing dbase)**
- **AR = 2, WR = 0, AW = 0, WW = 0**

```cpp
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();

    AccessDBase(ReadOnly); lock.Acquire();
    AR--; if (AR == 0 && WW > 0)
    okToWrite.signal();
    lock.Release();
}
```

Assume readers take a while to access database
Situation: Locks released, only AR is non-zero

Simulation of Readers/Writers Solution

- **R1 and R2 accessing dbase**
- **AR = 2, WR = 0, AW = 0, WW = 0**

```cpp
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();

    AccessDBase(ReadOnly); lock.Acquire();
    AR--; if (AR == 0 && WW > 0)
    okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- **W1 comes along (R1 and R2 are still accessing dbase)**
- **AR = 2, WR = 0, AW = 0, WW = 0**

```cpp
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; lock.release();

    AccessDBase(ReadWrite);
    lock.Acquire();
    AW--; if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- **W1 comes along (R1 and R2 are still accessing dbase)**
- **AR = 2, WR = 0, AW = 0, WW = 0**

```cpp
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; lock.release();

    AccessDBase(ReadWrite);
    lock.Acquire();
    AW--; if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```
Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing database)
  - AR = 2, WR = 0, AW = 0, WW = 1

```java
Writer() {
  lock.Acquire();
  while ((AW + AR) > 0) { // Is it safe to write?
    OKToWrite.wait(&lock); // No. Active users exist
    WW++; // Sleep on cond var
    No Longer waiting
  }
  AW++; lock.release();
  AccessDBase(ReadWrite);
  lock.Acquire();
  if (WW > 0) {
    OKToWrite.signal();
  } else if (WR > 0) {
    OKToRead.broadcast();
  }
  lock.Release();
}
```

Simulation of Readers/Writers Solution

- R3 comes along (R1 and R2 accessing database, W1 waiting)
  - AR = 2, WR = 0, AW = 0, WW = 1

```java
Reader() {
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();
  AccessDBase(ReadOnly);
  lock.Acquire();
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}
```

Simulation of Readers/Writers Solution

- R3 comes along (R1 and R2 accessing database, W1 waiting)
  - AR = 2, WR = 1, AW = 0, WW = 1

```java
Reader() {
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
  }
  AR++; // Now we are active!
  lock.release();
  AccessDBase(ReadOnly);
  lock.Acquire();
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}
Simulation of Readers/Writers Solution

- R3 comes along (R1, R2 accessing dbase, W1 waiting)
- AR = 2, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly); 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 Solution

- R1 and R2 accessing dbase, W1 and R3 waiting
- AR = 2, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
    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 Solution

- R2 finishes (R1 accessing dbase, W1 and R3 waiting)
- AR = 2, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
    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 Solution

- R2 finishes (R1 accessing dbase, W1 and R3 waiting)
- AR = 1, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
    lock.Acquire();
    AR--; if (AR == 0 && WW > 0)
    okToWrite.signal();
    lock.Release();
}
```
Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)
• AR = 1, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
    lock.Acquire();
    AR--;if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

R1 finishes (W1 and R3 waiting)
• AR = 1, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
    lock.Acquire();
    AR--;if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

R1 finishes (W1, R3 waiting)
• AR = 0, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
    lock.Acquire();
    AR--;if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```
Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
}
```

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

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```c
Writer() {
    lock.acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; lock.release();
    AccessDBase(ReadWrite);
}
```

```c
lock.acquire();
AW--; if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.Release();
```

Simulation of Readers/Writers Solution

- R1 signals a writer (W1 and R3 waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```c
Reader() {
    lock.acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
}
```

```c
lock.acquire();
AR--; if (AR == 0 && WW > 0)
okToWrite.signal();
lock.Release();
```

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```c
Writer() {
    lock.acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; lock.release();
    AccessDBase(ReadWrite);
}
```

```c
lock.acquire();
AW--; if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.Release();
```
Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 1, WW = 0

```java
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        okToWrite.wait(&lock); // No. Active users exist
        WW--;
    }
    AW++;
    lock.release();
    AccessDBase(ReadWrite);
    lock.Acquire();
    AW--; // No. Active users exist
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- W1 accessing dbase (R3 still waiting)
- AR = 0, WR = 1, AW = 1, WW = 0

```java
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        okToWrite.wait(&lock); // No. Active users exist
        WW--;
    }
    AW++;
    lock.release();
    AccessDBase(ReadWrite);
    lock.Acquire();
    AW--; // No. Active users exist
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- W1 finishes (R3 still waiting)
- AR = 0, WR = 1, AW = 1, WW = 0

```java
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        okToWrite.wait(&lock); // No. Active users exist
        WW--;
    }
    AW++;
    lock.release();
    AccessDBase(ReadWrite);
    lock.Acquire();
    AW--; // No. Active users exist
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- W1 finishes (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```java
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        okToWrite.wait(&lock); // No. Active users exist
        WW--;
    }
    AW++;
    lock.release();
    AccessDBase(ReadWrite);
    lock.Acquire();
    AW--; // No. Active users exist
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```
Simulation of Readers/Writers Solution

- W1 finishes (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

Writer() { 
    lock.Acquire();
    while ((AW + AR) > 0) {
        if (WW > 0)
            okToWrite.wait(&lock);
        else if (WR > 0)
            okToRead.broadcast();
    }
    AW++; lock.release();
    AccessDBase(ReadWrite);
}

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

Simulation of Readers/Writers Solution

- W1 signaling readers (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) {
        if (WR > 0)
            okToRead.wait(&lock);
        else if (AR > 0)
            okToWrite.signal();
    }
    AR++; lock.release();
    AccessDBase(ReadOnly);
}

lock.Acquire();
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();

Simulation of Readers/Writers Solution

- R3 gets signal (no waiting threads)
- AR = 0, WR = 1, AW = 0, WW = 0

Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) {
        if (WR > 0)
            okToRead.wait(&lock);
        else if (AR > 0)
            okToWrite.signal();
    }
    AR++; lock.release();
    AccessDBase(ReadOnly);
}

lock.Acquire();
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();

Simulation of Readers/Writers Solution

- R3 gets signal (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) {
        if (WR > 0)
            okToRead.wait(&lock);
        else if (AR > 0)
            okToWrite.signal();
    }
    AR++; lock.release();
    AccessDBase(ReadOnly);
}

lock.Acquire();
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();
Simulation of Readers/Writers Solution

- R3 accessing dbase (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
}
```

```c
lock.Acquire();
AR--;  // No longer active
if (AR == 0 && WW > 0) // No other active readers
    okToWrite.signal();
lock.Release();
```

Simulation of Readers/Writers Solution

- R3 finishes (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    AccessDBase(ReadOnly);
}
```

```c
lock.Acquire();
AR--;  // No longer active
if (AR == 0 && WW > 0) // No other active readers
    okToWrite.signal(); // Wake up one writer
lock.Release();
```

Questions

- Can readers starve? Consider Reader() entry code:

```c
while ((AW + WW) > 0) { // Is it safe to read?
    WR++; // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--; // No longer waiting
}
AR++; // Now we are active!
```

- What if we erase the condition check in Reader exit?

```c
AR--; // No longer active
```

- Further, what if we turn the signal() into broadcast()

```c
AR--; // No longer active
okToWrite.broadcast(); // Wake up sleepers
```

- Finally, what if we use only one condition variable (call it “okContinue”) instead of two separate ones?

  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()
Use of Single CV: okContinue

Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okContinue.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    if ((AW > 0) || (AR > 0)) {
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}

Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        AW++;
        okContinue.wait(&lock);
        AW--;
    }
    AR--;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    if (WW > 0) {
        okContinue.signal();
    } else if (WR > 0) {
        okContinue.broadcast();
    }
    lock.Release();
}

What if we turn okToWrite and okToRead into okContinue (i.e. use only one condition variable instead of two)?

Consider this scenario:
- R1 arrives
- W1, R2 arrive while R1 still reading → W1 and R2 wait for R1 to finish
- Assume R1’s signal is delivered to R2 (not W1)

Need to change to broadcast()!

Must broadcast() to sort things out!

Use of Single CV: okContinue

Reader() {
// check into system
lock.Acquire();
while ((AW + WW) > 0) {
    WR++;
    okContinue.wait(&lock);
    WR--;
} AR++;
lock.release();
// read-only access
AccessDbase(ReadOnly);
// check out of system
lock.Acquire();
if (AR == 0 & WW > 0) okContinue.signal();
lock.Release();
}

Writer() {
// check into system
lock.Acquire();
while ((AW + AR) > 0) {
    AW++;
    okContinue.wait(&lock);
    AW--;
} AR--;
lock.release();
// read/write access
AccessDbase(ReadWrite);
// check out of system
lock.Acquire();
if (WW > 0 || WR > 0) okContinue.broadcast();
lock.Release();
}

Can we construct Monitors from Semaphores?
• Locking aspect is easy: Just use a mutex
• Can we implement condition variables this way?
  Wait() { semaphore.P(); }
  Signal() { semaphore.V(); }

• Does this work better?
  Wait(Lock lock) {
      lock.Release();
      semaphore.P();
      lock.Acquire();
  }
  Signal() { semaphore.V(); }

  Need to change to broadcast()!

  Must broadcast() to sort things out!
Construction of Monitors from Semaphores (con’t)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative

- Does this fix the problem?

```c
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() {
    if semaphore queue is not empty
        semaphore.V();
}
```

- Not legal to look at contents of semaphore queue
- There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()

- It is actually possible to do this correctly
- Complex solution for Hoare scheduling in book
- Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed

- Basic structure of monitor-based program:

```c
lock
while (need to wait) {
    condvar.wait();
} unlock
```

- Check and/or update state variables
- Wait if necessary

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know all the code paths out of a critical section

```c
int Rtn() {
    lock.acquire();
    ...if (exception) {
        lock.release();
        return errReturnCode;
    }
    ...lock.release();
    return OK;
}
```

- Watch out for `setjmp/longjmp`!
  - Can cause a non-local jump out of procedure
  - In example, procedure E calls longjmp, popping stack back to procedure B
  - If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)

```c
void Rtn() {
    lock.acquire();
    ...DoFoo();
    ...lock.release();
} void DoFoo() {
    ...if (exception) throw errException;
    ...
}
```

- Notice that an exception in DoFoo() will exit without releasing the lock!
**C++ Language Support for Synchronization (con't)**

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:
    ```
    void Rtn() {
        lock.acquire();
        try {
            ...DoFoo();
        } catch (...) { // catch exception
            lock.release(); // release lock
            throw; // re-throw the exception
        }
        lock.release();
    }
    void DoFoo() {
        if (exception) throw errException;
    }
    ```

  - Can deallocate/free lock regardless of exit method

---

**Java Language Support for Synchronization**

- Java has explicit support for threads and thread synchronization
- Bank Account example:
  ```
  class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
  }
  ```
  - Every object has an associated lock which gets automatically acquired and released on entry and exit from a synchronized method.

---

**Java Language Support for Synchronization (con't)**

- Java also has synchronized statements:
  ```
  synchronized (object) {
    ...
  }
  ```
  - Since every Java object has an associated lock, this type of statement acquires and releases the object’s lock on entry and exit of the body
  - Works properly even with exceptions:
    ```
    synchronized (object) {
        ...DoFoo();
        ...}
    void DoFoo() {
        throw errException;
    }
    ```

---

**Java Language Support for Synchronization (con't 2)**

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method of block:
    ```
    void wait(long timeout); // Wait for timeout
    void wait(long timeout, int nanoseconds); //variant
    void wait();
    ```
  - How to signal in a synchronized method or block:
    ```
    void notify(); // wakes up oldest waiter
    void notifyAll(); // like broadcast, wakes everyone
    ```
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:
    ```
    t1 = time.now();
    while (!ATMRequest()) {
        wait (CHECKPERIOD);
        t2 = time.new();
        if (t2 - t1 > LONG_TIME) checkMachine();
    }
    ```
  - Not all Java VMs equivalent!
    ```
    » Different scheduling policies, not necessarily preemptive!
    ```
Summary

- **Semaphores**: Like integers with restricted interface
  - 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**: A 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()`
- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed