Synchronization (cont’d), Readers/Writers example

September 24th, 2017
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
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
Simple Monitor Example (version 1)

• Here is an (infinite) synchronized queue

```java
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!
Condition Variables

• How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  – Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone

• Condition Variable: a queue of threads waiting for something inside a critical section
  – Key idea: allow sleeping 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)
Complete Monitor Example (with cond. variable)

• Here is an (infinite) synchronized queue

```java
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:

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

– Why didn’t we do this?

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

• Answer: depends on the type of scheduling
  – Hoare-style
  – Mesa-style
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

```java
... lock.Acquire();
... dataready.signal();
... lock.Release();

lock.Acquire()
... if (queue.isEmpty()) {
  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();
...
put waiting thread on ready queue

dataready.signal();
...
lock.Release();

lock.Acquire();
...
while (queue.isEmpty()) {
    dataready.wait(&lock);
}
...
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();
}

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

Replace “while” with “if”
Mesa Monitor: Why “while()”?

RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
Mesa Monitor: Why “while()”?

App. Shared State

```
queue
```

Monitor

- lock: BUSY (T1)
- dataready
- queue → NULL

CPU State

- Running: T1
- Ready
- queue → NULL
- ...

T1 (Running)

```
RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
```
Mesa Monitor: Why “while()”?

API. Shared State

queue

Monitor

lock: FREE
dataready queue

CPU State

Running:
Ready
queue \rightarrow\text{NULL}

T1 \text{(Waiting)}

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

wait(\&\text{lock})\text{ puts thread on dataready queue and releases lock}
Mesa Monitor: Why “while()”?

App. Shared State

queue

Monitor

lock: FREE
dataready

queue → T1

CPU State

Running: T2
Ready
queue → NULL
...

T1 (Waiting)

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

T2 (Running)

AddToQueue(item) {
lock.Acquire();
queue.enqueue(item);
dataready.signal();
lock.Release();
}
Mesa Monitor: Why “while()”?

App. Shared State

- `queue` add item

Monitor

- `lock: BUSY (T2)`
- `dataready queue` → T1

CPU State

- Running: T2
- Ready
- `queue` → NULL

T1 (Waiting)

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

T2 (Running)

```java
AddToQueue(item) {
    lock.Acquire();
    queue.enqueue(item);
    dataready.signal();
    lock.Release();
}
```
Mesa Monitor: Why “while()”?

T1 (Ready)
RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}

T2 (Running)
AddToQueue(item) {
    lock.Acquire();
    queue.enqueue(item);
    dataready.signal();
    lock.Release();
}

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

App. Shared State

```
queue
```

Monitor

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

CPU State

Running: T2
Ready
queue → T1, T3 ...

T1 (Ready)

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

T2 (Running)

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

T3 (Ready)

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```
Mesa Monitor: Why “while()”?

App. Shared State

| queue |

Monitor

lock: FREE

dataready

queue → NULL

CPU State

Running:

Ready

queue → T1, T3

...

T1 (Ready)

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

T2 (Terminate)

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

T3 (Ready)

RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
Mesa Monitor: Why “while()”?

App. Shared State

queue

Monitor

lock: FREE

dataready.wait(&lock);

item = queue.dequeue();
lock.Release();
return(item);

CPU State

Running: T3

Ready

queue \rightarrow T1

T3 scheduled first!

T3 (Running)

RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
Mesa Monitor: Why “while()”?

App. Shared State

Monitor

CPU State

T1 (Ready)

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

T3 (Running)

RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
Mesa Monitor: Why “while()”?

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

T1 (Ready)

T3 (Running)

RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
Mesa Monitor: Why “while()”?

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

**T1 (Ready)**

**T3 (Finished)**

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 (T1)</td>
<td>Running: T1</td>
</tr>
<tr>
<td></td>
<td>dataready</td>
<td>Ready</td>
</tr>
<tr>
<td></td>
<td>queue ← NULL</td>
<td>queue ← NULL</td>
</tr>
</tbody>
</table>

**T1 (Running)**

```java
RemoveFromQueue() {
    lock.Acquire();
    if (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item = queue.dequeue();
    lock.Release();
    return(item);
}
```
Mesa Monitor: Why “while()”?

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

T1 (Running)

ERROR: Nothing in the queue!
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”
Mesa Monitor: Why “while()”?

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

T1 (Ready)

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

App. Shared State

```
queue
```

Monitor

```
lock: FREE

dataready

queue → T1
```

CPU State

```
Running: T1
Ready
queue → NULL
...
```

T1 (*Waiting*)

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

• Midterm on Monday 10/1 5:30-6PM

• Closed book, no calculators, one double-side letter-sized page of handwritten notes
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

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();
   AR--; // No longer active
   if (AR == 0 && WW > 0) // No other active readers
      okToWrite.signal(); // Wake up one writer
   lock.Release();
}
Code for a Writer

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();
    AW--;                   // No longer active
    if (WW > 0) {           // Give priority to writers
        okToWrite.signal(); // Wake up one writer
    } else if (WR > 0) {    // Otherwise, wake reader
        okToRead.broadcast(); // Wake all readers
    }

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

- R1 comes along
- AR = 0, 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

• R1 comes along
• 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

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

```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();
}
```

```java
AccessDbase(ReadOnly);
```

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

- R1 comes along
- 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++;
    lock.release(); // Now we are active!
}

AccessDbase(ReadOnly)

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

- R2 comes along
- 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
- 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);

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

- R2 comes along
- 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--; // Now we are active!
    if (AR == 0 && WW > 0) { // No longer waiting
        okToWrite.signal();
    }
    lock.Release();
}
Simulation of Readers/Writers Solution

• R2 comes along
• AR = 2, 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);

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

- R2 comes along
- 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--;
    }
    AR++;
    // Now we are active!
    lock.release();
    AccessDbase(ReadOnly);
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
}
```

Assume readers take a while to access database

Situation: Locks released, only AR is non-zero
Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, 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--; // Sleep on cond var
    }
    AW++;
    lock.release();

    AccessDbase(ReadWrite);

    lock.Acquire();
    AW--; // No longer waiting
    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

```java
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) {
        // Is it safe to write?
        // No. Active users exist
        WW++;
        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 = 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--;
  }
  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 = 1

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

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

W1 cannot start because of readers, so goes to sleep
Simulation of Readers/Writers Solution

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

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

- R3 comes along (R1, R2 accessing dbase, 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();
}
```

```java
AccessDbase(ReadOnly);
```

```java
lock.Acquire();
AR--;
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

Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) {
        // Is it safe to read?
        WR++;
        // No. Writers exist
        okToRead.wait(&lock);
        // Sleep on cond var
        WR--;
    }
    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

- R3 comes along (R1, R2 accessing dbase, 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();
AR--;```

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, R3 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();
AR--; // is not safe to write
if (AR == 0 && WW > 0) {
    okToWrite.signal();
    lock.Release();
}
```
Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, R3 waiting)
- AR = 1, 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();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```
Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, R3 waiting)
- AR = 1, 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();
    AR--;
    if (AR == 0 && WW > 0) {
        okToWrite.signal();
    }
    lock.Release();
}
```
Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, 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--; // Now we are active!
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}
```
Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- AR = 1, 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(); // lock.Acquire();
    AR--; // AR--;
    if (AR == 0 && WW > 0) // 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

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--; // This line is highlighted
    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

```java
Reader() {
    lock.Acquire();
    while (((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Wait 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

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

All reader finished, signal writer – note, R3 still waiting
Simulation of Readers/Writers Solution

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

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

    lock.Acquire();
    AW--; // Is it safe to write?
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

Got signal from R1
Simulation of Readers/Writers Solution

- W1 gets signal (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
        // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    lock.release();

    AccessDbase(ReadWrite);
}

lock.Acquire();
AW--; // Is it safe to write?
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--; // 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 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);
```

```java
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 = 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--;
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 = 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--;
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 = 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--;
    } // No longer waiting
    AW++;
    lock.release();

    AccessDbase(ReadWrite);

    lock.Acquire();
    AW--; // No waiting writer, signal reader R3
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

No waiting writer, signal reader R3
Simulation of Readers/Writers Solution

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

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

- R1 finishes (W1, R3 waiting)
- 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--;
    }
    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 = 0, AW = 0, WW = 0

```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)
```

```java
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 = 0, AW = 0, WW = 0

```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();
AR--; // Now we are active!
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();
```
Simulation of Readers/Writers Solution

• R1 finishes (W1, R3 waiting)
• AR = 0, 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();
}
DONE!
```
Read/Writer Questions

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

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.Release();
}

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

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
Read/Writer Questions

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

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToRead.broadcast();
    lock.Release();
}

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

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0)
        okToWrite.signal();
    else if (WR > 0)
        okToRead.broadcast();
    lock.Release();
}

What if we turn signal to broadcast?
What if we turn `okToWrite` and `okToRead` into `okContinue`?
Read/Writer Questions

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

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

    // read/write access
    AccessDbase(ReadWrite);

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

• 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)
Read/Writer Questions

**Reader()**

```c
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();
    AR--;  
    if (AR == 0 && WW > 0)  
        okContinue.broadcast();
    lock.Release();
}
```

**Writer()**

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

    // read/write access
    AccessDbase(ReadWrite);

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

Need to change to broadcast!
Synchronization Summary

• **Monitors**: A lock plus zero or more condition variables
  – Always acquire lock before accessing shared data
  – Use condition variables to wait inside critical section
    » Three Operations: `Wait()`, `Signal()`, `Broadcast()`