CSE150
Operating Systems
Lecture 9

Readers-Writers Problem, Language Support for Concurrent Programming

Projects

- Write sudo code in your design report, but not copy your c or java code.
- Test your project code by: provided test cases.
- Write your own test cases. It will improve your report a lot.
  - Java
  - C, then convert to .coff

Review: Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer

- Example: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty

Review: Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffer, if none (scheduling constraint)
  - Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)

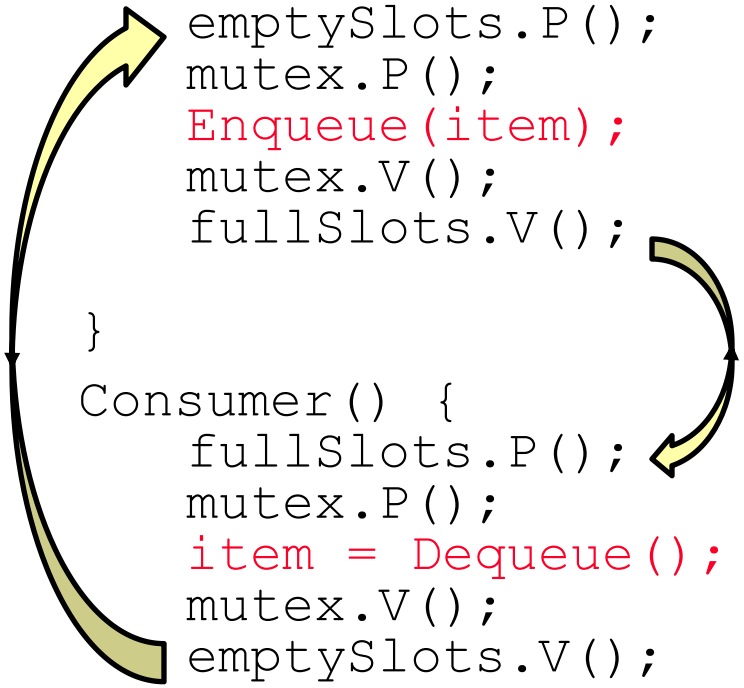
Review: Correctness constraints for solution

- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullSlots; // consumer's constraint`
  - `Semaphore emptySlots;// producer's constraint`
  - `Semaphore mutex;     // mutual exclusion`

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;
}
```



Review: Monitor

- Semaphores are confusing because dual purpose:
  - Both mutual exclusion and scheduling constraints
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

Review: Definition of Monitor

- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
- Lock: provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
- 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

Review: Condition Variables

- Condition variable: a variable `x` that implements:
  - `x.wait()`: Wait on a condition (go to sleep?)
  - `x.signal()`: Wake up one waiter, if any
  - Many threads can call `x.wait()`, they will be queued up waiting for a call to `x.signal()`. That call will start the first waiting thread.

- To support sleeping while waiting inside critical section, we add
  - `x.wait(&lock)`: Atomically release the lock and go to sleep. Re-acquire lock later, before returning.
- Some systems also implement:
  - `broadcast()` to wake up all waiting threads

- Rule: Must hold lock when doing condition variable operations

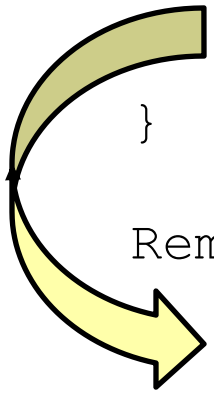
Complete Monitor Example (condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;                        // shared data


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);
}
```



Today

- Monitors
- Language Support for Synchronization

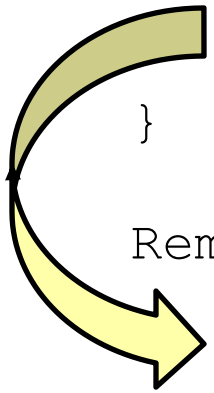
Complete Monitor Example (condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;                        // shared data


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);
}
```



Why "while"? But not "if"?

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

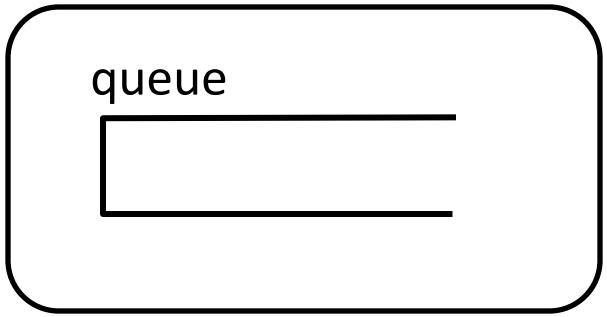
```
while (queue.isEmpty()) {
  dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();// Get next item
```

  - Why didn't we do this?

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

Coke machine example

App. Shared State



Monitor

lock: FREE

datready queue → NULL

CPU State

Running: T1

Ready queue → NULL

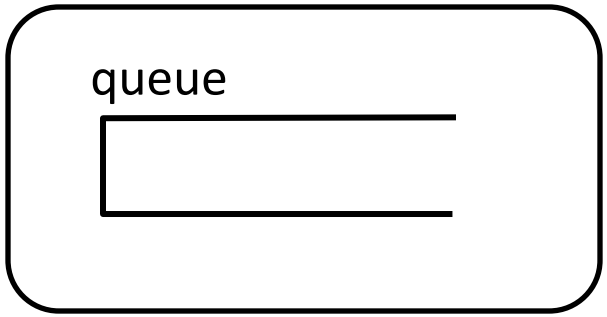
…

T1 (Running)

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

Coke machine example

App. Shared State



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);
}
```

Coke machine example

App. Shared State

queue

Monitor

lock: FREE

dataready queue → T1

CPU State

Running:

Ready queue → NULL

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

T1 (Waiting)

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

Coke machine example

App. Shared State

queue

Monitor

lock: FREE

datready queue → T1

CPU State

Running: T2

Ready queue → NULL

…

T1 (Waiting)

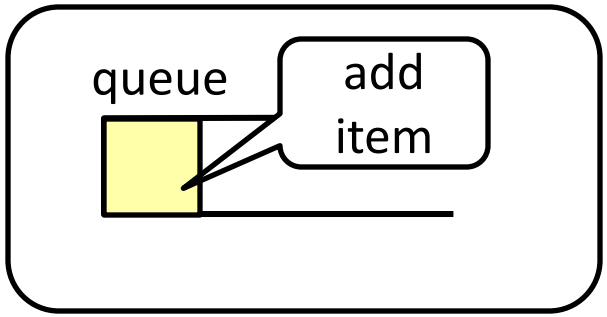
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

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

Coke machine example

App. Shared State



Monitor

lock: BUSY (T2)

dataready queue → T1

CPU State

Running: T2

Ready queue → NULL

…

T1 (Waiting)

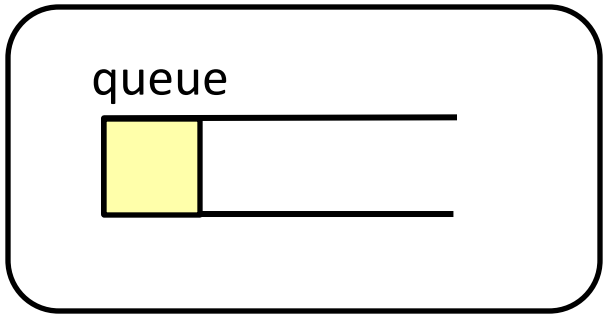
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T2 (Running)

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

Coke machine example

App. Shared State



Monitor

lock: BUSY (T2)

dataready queue → NULL

CPU State

Running: T2

Ready queue → T1

…

T1 (Ready)

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

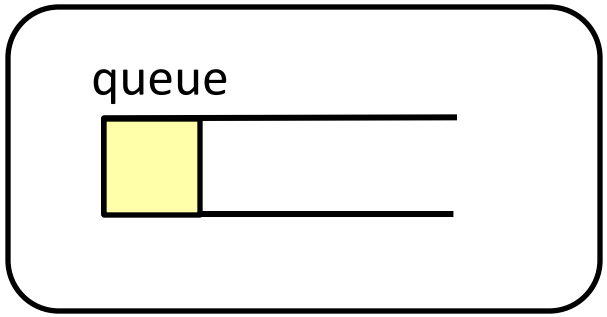
T2 (Running)

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

signal() wakes up T1 and moves it on ready queue

Coke machine example

App. Shared State



Monitor

lock: BUSY (T2)

dataready queue → NULL

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)

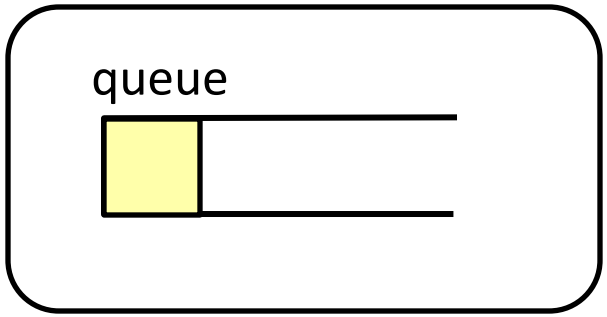
```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

T3 (Ready)

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

Coke machine example

App. Shared State



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)

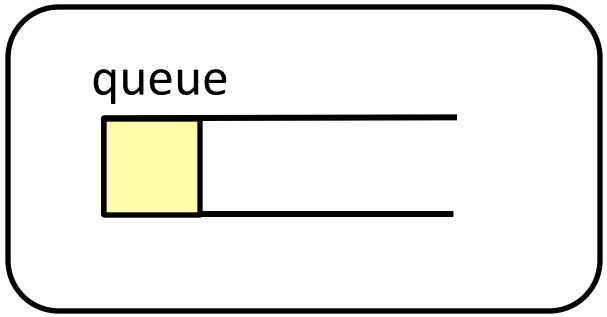
```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

T3 (Ready)

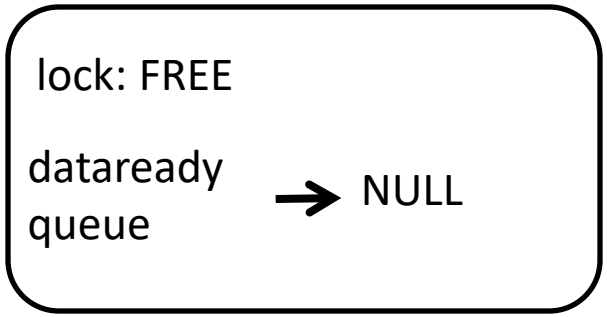
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

Coke machine example

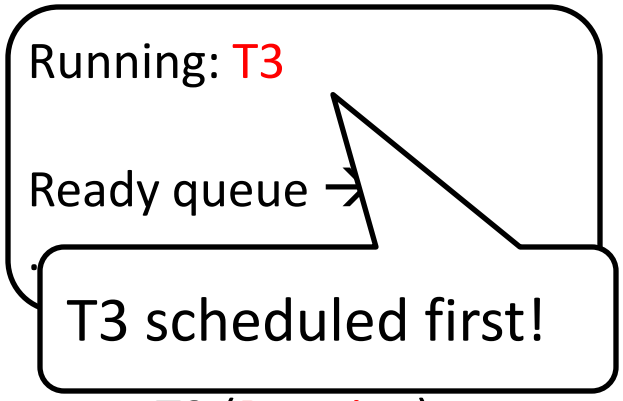
App. Shared State



Monitor



CPU State



T1 (Ready)

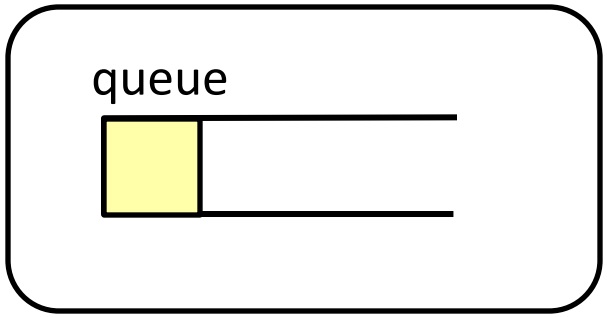
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

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

Coke machine example

App. Shared State



Monitor

lock: BUSY (T3)

dataready queue → NULL

CPU State

Running: T3

Ready queue → T1

…

T1 (Ready)

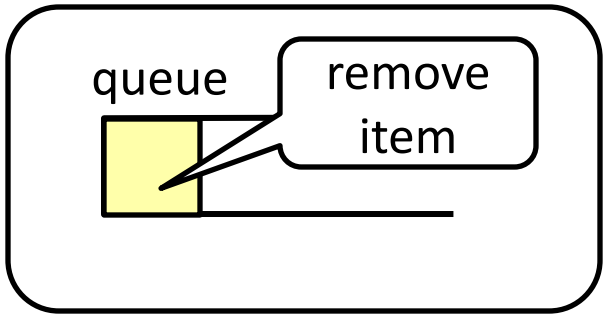
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

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

Coke machine example

App. Shared State



Monitor

lock: BUSY (T3)

dataready queue → NULL

CPU State

Running: T3

Ready queue → T1

…

T1 (Ready)

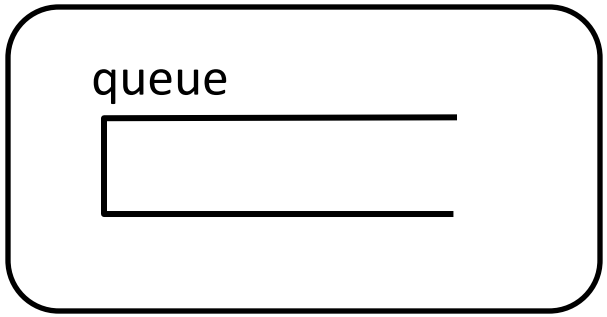
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Running)

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

Coke machine example

App. Shared State



Monitor

lock: FREE

dataready queue → NULL

CPU State

Running:

Ready queue → T1

…

T1 (Ready)

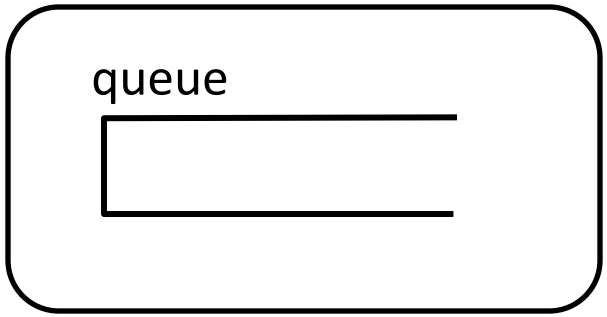
```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

T3 (Finished)

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

Coke machine example

App. Shared State



Monitor

lock: BUSY (T1)

dataready queue → NULL

CPU State

Running: T1

Ready queue → NULL

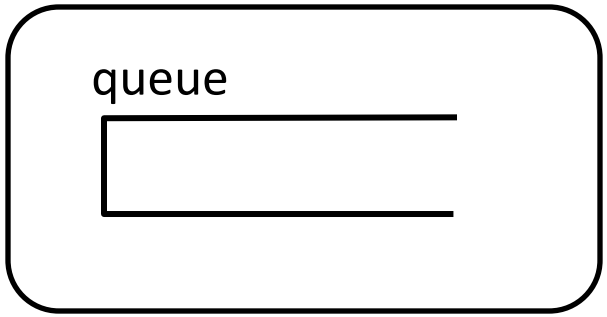
…

T1 (Running)

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

Coke machine example

App. Shared State



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);
}
```

NULL: Nothing in the queue!

Coke machine example

App. Shared State

queue

Monitor

lock: BUSY (T1)

dataready queue → NULL

CPU State

Running: T1

Ready queue → NULL

…

T1 (Running)

```
RemoveFromQueue() {
  lock.Acquire();
  while(queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = que
  lock.Relea
  return(item
}
```

Replace "if" with "while"

Coke machine example

App. Shared State

queue

Monitor

lock: BUSY (T1)

dataready queue → NULL

CPU State

Running: T1

Ready queue → NULL

…

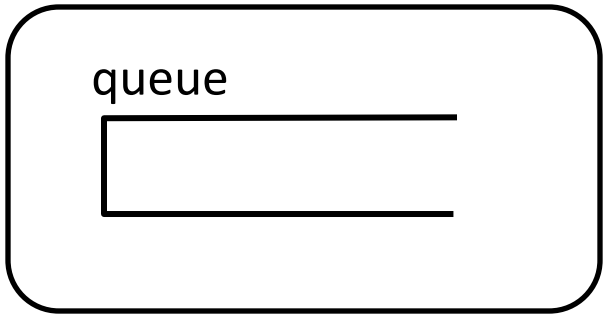
T1 (Ready)

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

Check again if empty!

Coke machine example

App. Shared State



Monitor

lock: FREE

dataready queue → T1

CPU State

Running: T1

Ready queue → NULL

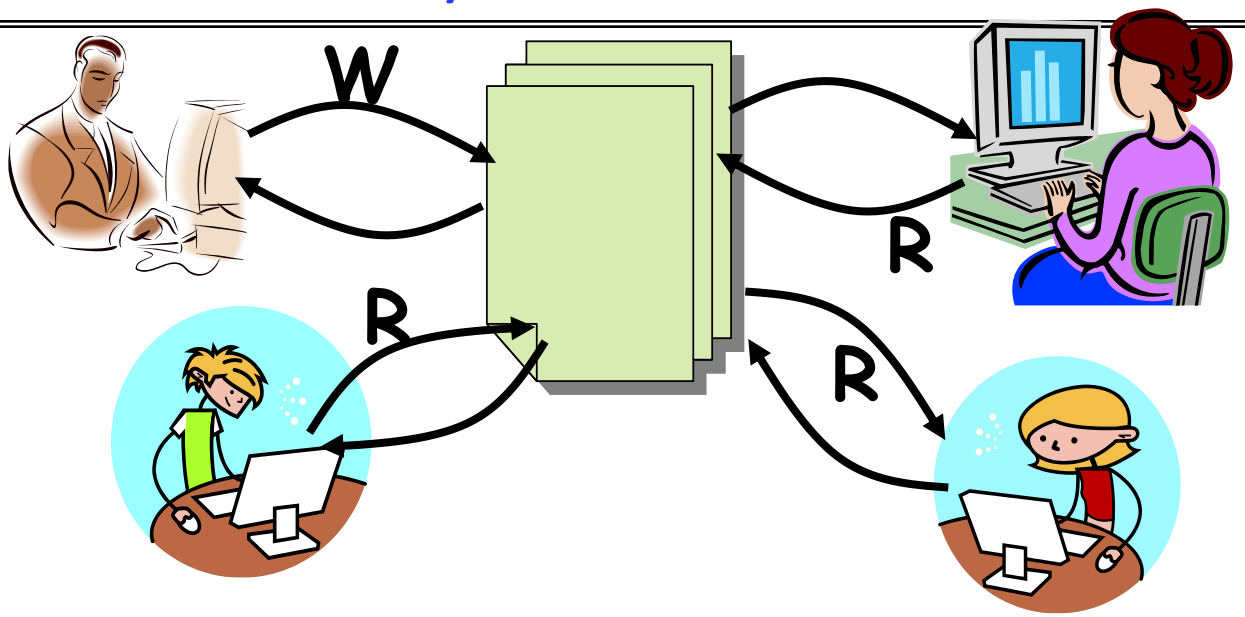
…

T1 (Waiting)

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

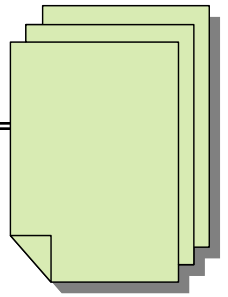
http://research.microsoft.com/en-us/um/people/blampson/23-processesinmesa/Acrobat.pdf

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



- 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
    - » Conditioin okToWrite = NIL
- 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

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--;
    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

```
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

- 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

```
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

```
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

```
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

```
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

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

```
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

```
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 = 1

```
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 = 1

```
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();
}
```

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

```
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 = 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--;
    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--;                  // 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

```
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

```
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

```
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

```
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 = 1, 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--;
    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--;
    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--;
    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--;
    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

```
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();
  }
```

Got signal from R1

Simulation of Readers/Writers Solution

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

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

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

```
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 accessing database (R3 still waiting)
- AR = 0, WR = 1, AW = 1, WW = 0

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

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

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

```
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();
}
```

No waiting writer, signal reader R3

Simulation of Readers/Writers Solution

- R3 gets signal
- AR = 0, WR = 1, 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();
  }
```

Got signal from W1

Simulation of Readers/Writers Solution

- R3 gets signal
- 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

- R3 gets signal (No waiters)
- 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

- R3 accesses database
- 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

- R3 accesses database
- 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

- R3 finishes
- 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

- R3 finishes
- 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!

Reader/Writer Questions

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

    // read-only
    AccessDbase

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

What if we remove this line?

```
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();
}
```

Reader/Writer Questions

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

    // read-only
    AccessDbase(

    // check out
    lock.Acquire
    AR--;
    if (AR == 0 && WW > 0)
       okToWrite.broadcast();
    lock.release();
}
```

What if we turn signal to broadcast?

```
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();
}
```

Reader/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/writ
    AccessDbase

    // check out
    lock.Acquire
    AW--;
    if (WW > 0){
       okToWrite.sig al();
    } else if (WR > 0) {
       okToRead.signal();
    }
    lock.release();
  }
```

What if we turn broadcast to signal? starvation

Reader/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();
    }
    lock.release();
}
```

What if we turn okToWrite and okToRead into okContinue?

Reader/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();
    }
    lock.release();
}
```

- **W1, R2 arrive while R1 still reading → W1 and R2 wait for R1 to finish**
- **R1 signals R2 (instead of W1)**
- **R2 will immediately go to sleep (because of WW)…and threads will only be woken up if a new thread comes along**

Reader/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.broadcast();
    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.broadcast();
    } else if (WR > 0) {
       okContinue.broadcast();
    }
    lock.release();
}
```

Need to change to broadcast!

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
  - Can be implemented by semaphores.
- Basic structure of monitor-based program:

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

do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables
Wait if necessary

Check and/or update state variables

Java Language Support for Synchronization

Java supports both low-level and high-level synchronization:

- Low-level:
  - Lock class: a lock, with methods:
    - » lock.lock()
    - » lock.unlock()
  - Condition: a condition variable associated with a lock, methods:
    - » condvar.await()
    - » condvar.signal()

- High-level: every object has an ***implicit*** lock and condition variable(s)
  - synchronized keyword, applies to methods or blocks
  - Implicit condition variable methods:
    - » wait()
    - » notify() and notifyAll()

Java Language Low-level Synchronization

```
public class SynchronizedQueue {
    private Lock lock = new ReentrantLock();
    private Condition cv = lock.newCondition();
    private LinkedList<Integer> q
                          = new LinkedList<Integer>();

    public void enqueue(int item) {
        try {
            lock.lock();
            q.add(item);
            cv.signal();
        } finally {
            lock.unlock();
        }
    }
```

Summary

- Reader/Writer Solution
  - More flexible than you think
  - Modifications at the cost of efficiency