Goals for Today

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Goals for Today

- Continue with Synchronization Abstractions
  - Semaphores and monitors

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Goals for Today

- Continue with Synchronization Abstractions
  - Semaphores and monitors
- Readers-Writers problem and solutoin

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Goals for Today

- Continue with Synchronization Abstractions
  - Semaphores and monitors
- Readers-Writers problem and solutoin
- Language Support for Synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Recall: Semaphores

Recall: Semaphores

- **Definition: a Semaphore has a non-negative integer value and supports the following two operations:**

Recall: 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**

Recall: Semaphores

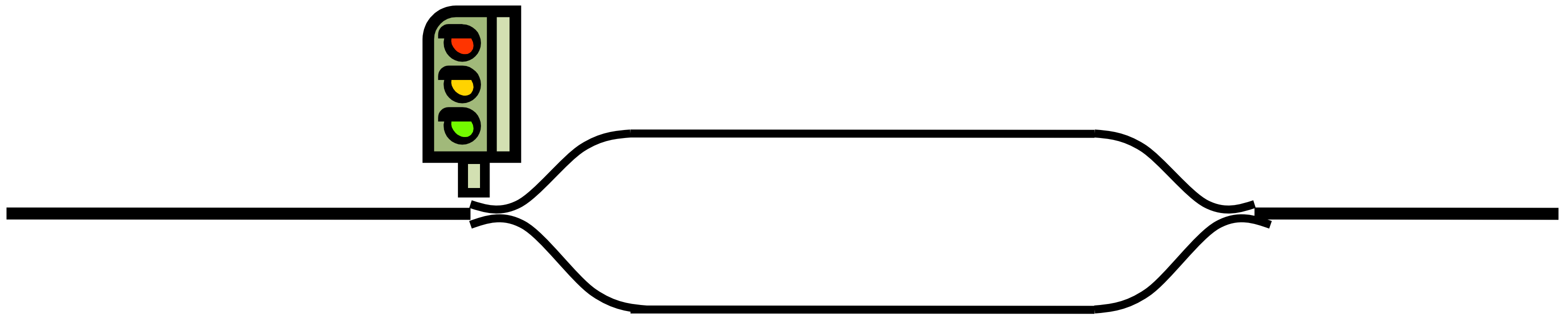
- **Definition: a Semaphore has a non-negative integer value and supports the following two operations:**
  - **P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1**
    - » Think of this as the wait() operation
  - **V(): an atomic operation that increments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

Recall: 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

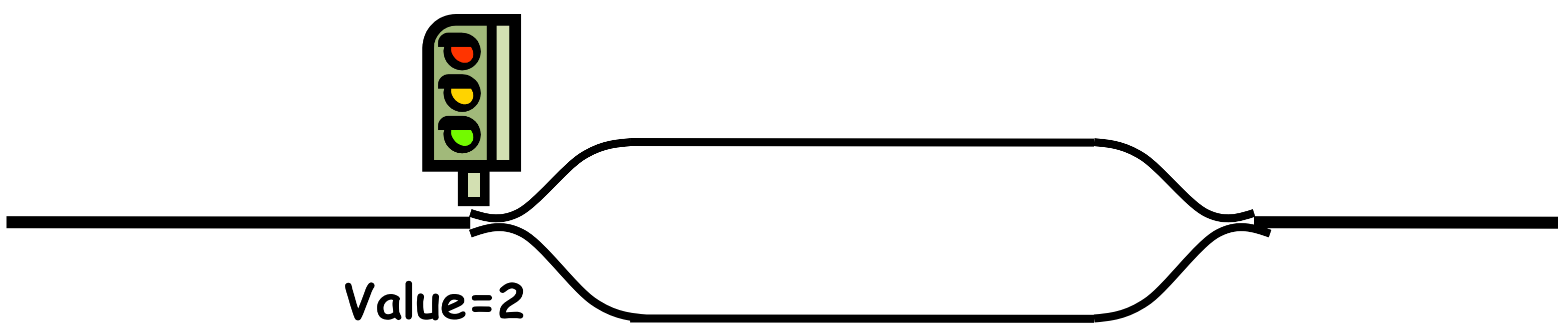
Recall: Semaphores

- 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



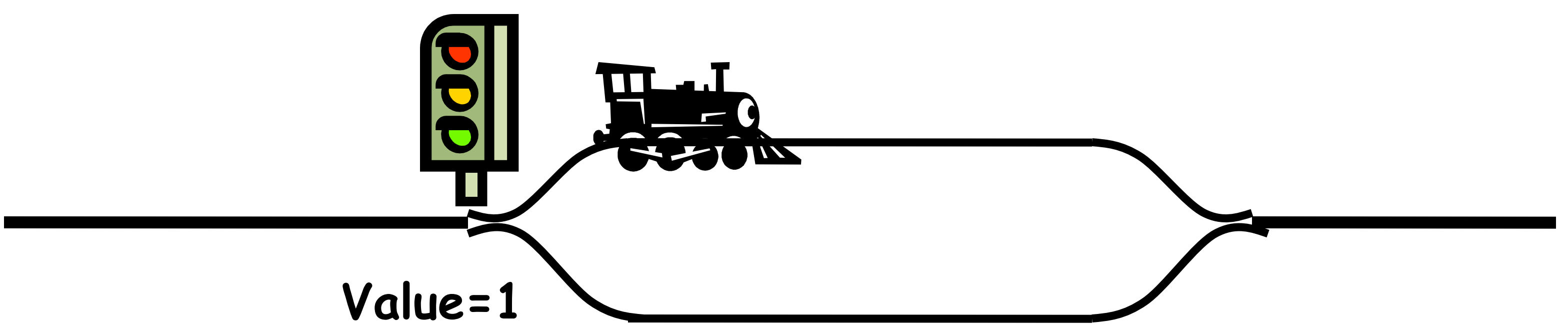
Recall: Semaphores

- 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



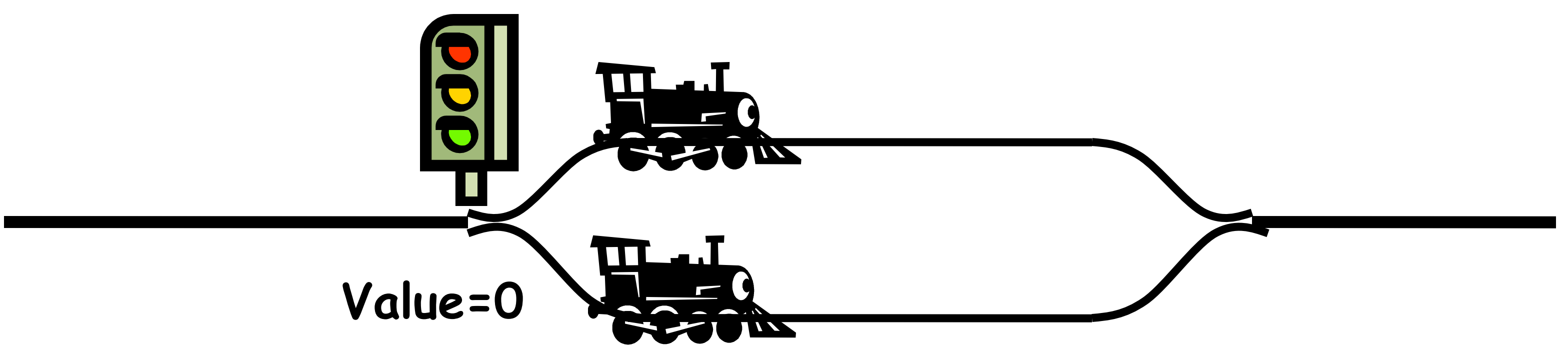
Recall: Semaphores

- 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



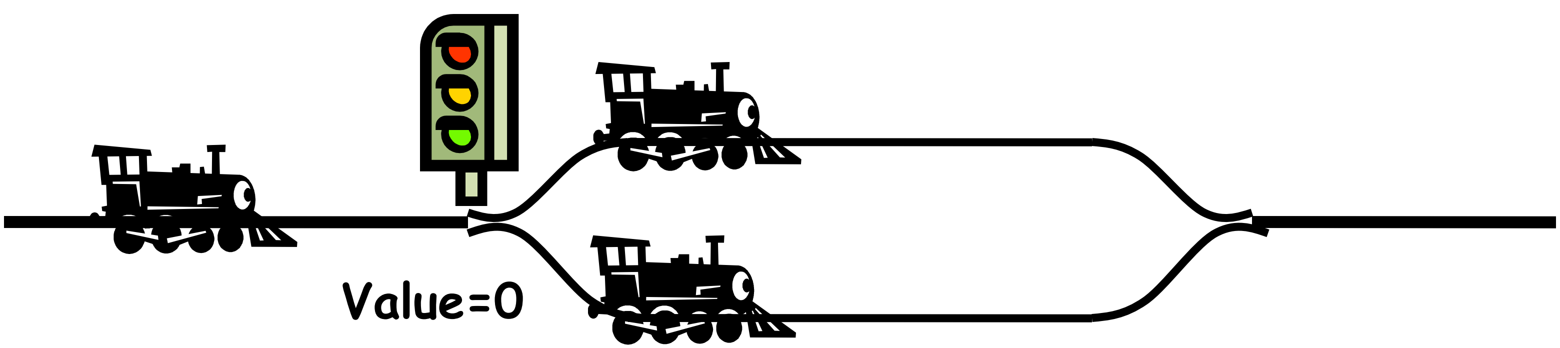
Recall: Semaphores

- 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



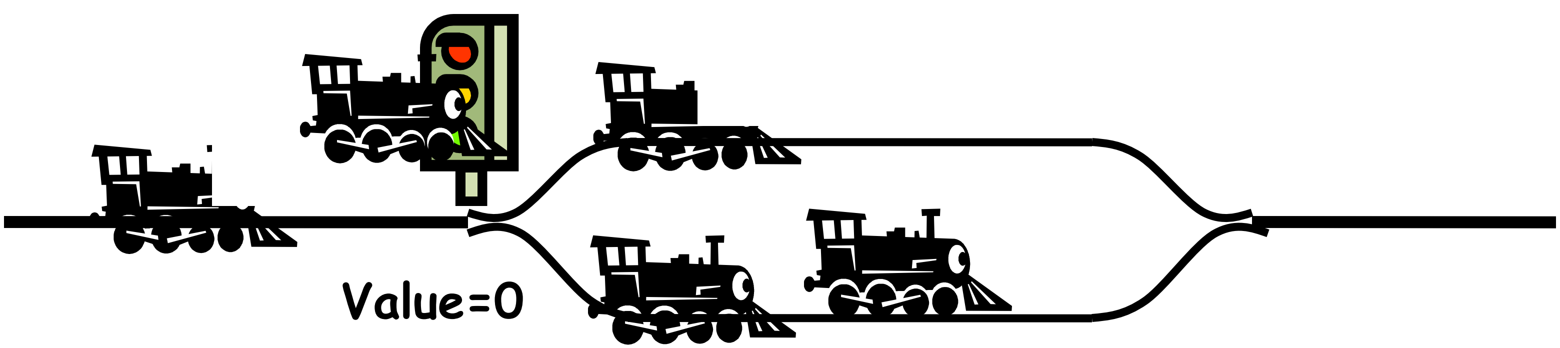
Recall: Semaphores

- 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



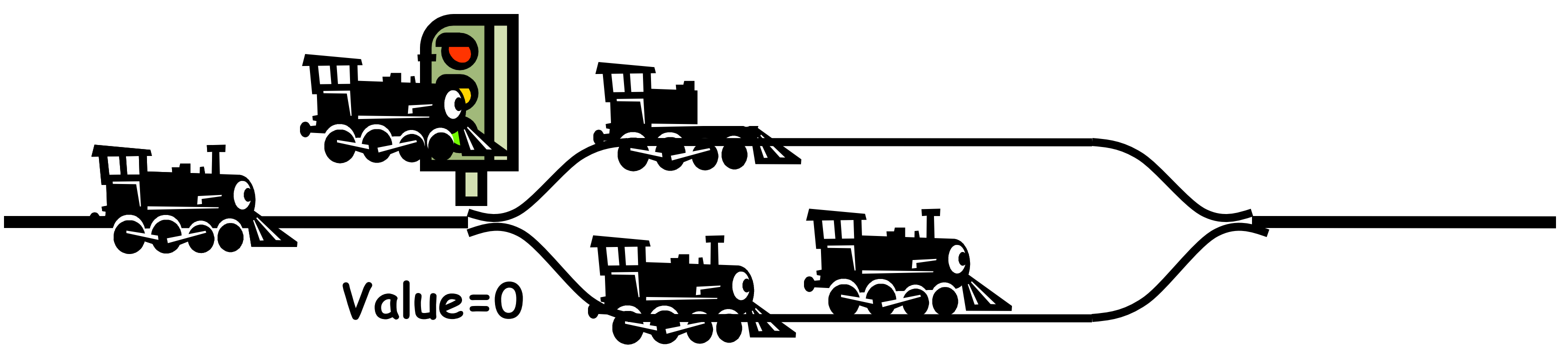
Recall: Semaphores

- 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



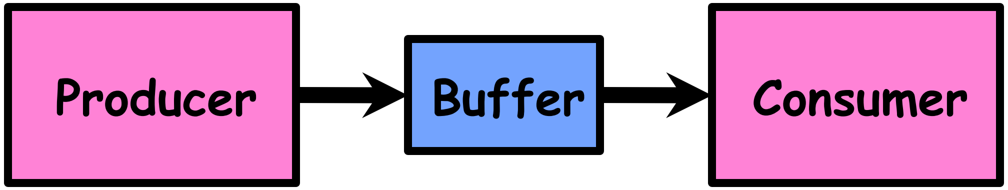
Recall: Semaphores

- 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



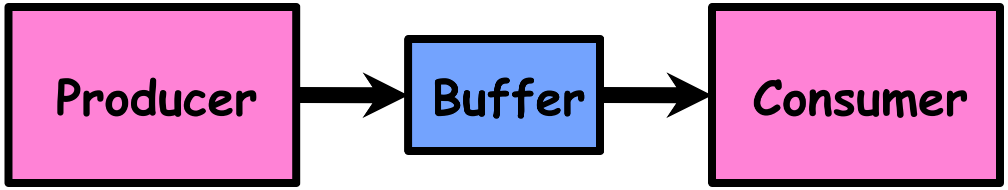
Producer-consumer with a bounded buffer

Producer-consumer with a bounded buffer



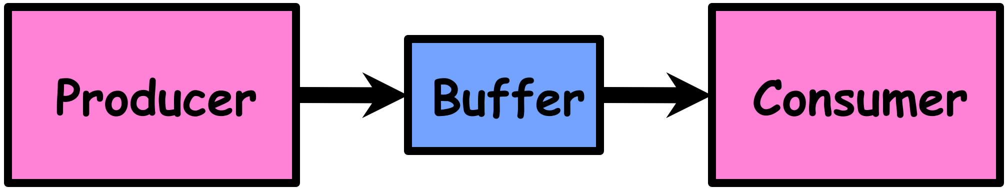
- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out

Producer-consumer with a bounded buffer



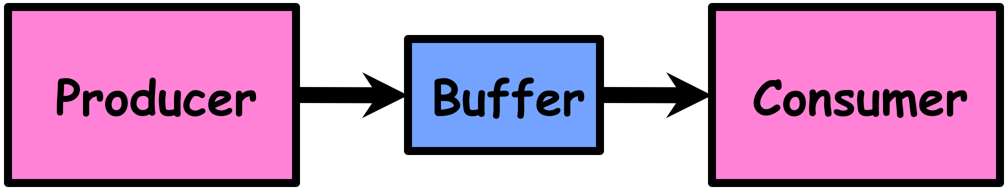
- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out

Producer-consumer with a bounded buffer



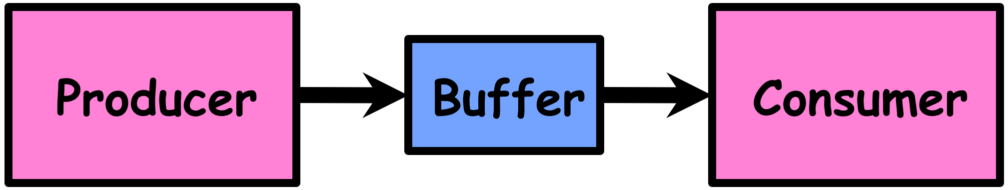
- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

Producer-consumer with a bounded buffer



- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

Producer-consumer with a bounded buffer



- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out

- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

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

Correctness constraints for solution

Correctness constraints for solution

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

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;       // mutual exclusion`

Full Solution to Bounded Buffer (Coke Machine)

Full Solution to Bounded Buffer (Coke Machine)

```
Semaphore fullBuffer = 0; // Initially, no coke
```

Full Solution to Bounded Buffer (Coke Machine)

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                             // Initially, num empty slots
```

Full Solution to Bounded Buffer (Coke Machine)

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine
```

Full Solution to Bounded Buffer (Coke Machine)

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                             // Initially, num empty slots
Semaphore mutex = 1;         // No one using machine

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

Full Solution to Bounded Buffer (Coke Machine)

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

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

Consumer() {
    fullBuffers.P();      // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();     // tell producer need more
    return item;
}
```

Discussion about Solution

Discussion about Solution

- **Why asymmetry?**

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`

Discussion about Solution

- **Why asymmetry?**
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P()`, `fullBuffer.V()`
  - Consumer does: `fullBuffer.P()`, `emptyBuffer.V()`
- Is order of P's important?

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
  - Yes! Can cause deadlock

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P()`, `fullBuffer.V()`
  - Consumer does: `fullBuffer.P()`, `emptyBuffer.V()`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

Motivation for Monitors and Condition Variables

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - » Both mutual exclusion and scheduling constraints
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - » Both mutual exclusion and scheduling constraints
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - 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
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - » Both mutual exclusion and scheduling constraints
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - 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
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language
- The lock provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data

Critical Section: Monitors

- Basic idea:
  - Restrict programming model
  - Permit access to shared variables only within a critical section
- General program structure
  - Entry section
    - “Lock” before entering critical section
    - Wait if already locked
    - Key point: synchronization may involve wait
  - Critical section code
  - Exit section
    - “Unlock” when leaving the critical section
- Object-oriented programming style
  - Associate a lock with each shared object
  - Methods that access shared object are critical sections
  - Acquire/release locks when entering/exiting a method that defines a critical section

Simple Monitor Example (version 1)

Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

Simple Monitor Example (version 1)

- **Here is an (infinite) synchronized queue**

```
Lock lock;
Queue queue;
```

Simple Monitor Example (version 1)

- **Here is an (infinite) synchronized queue**

```
Lock lock;
Queue queue;
```

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
}
```

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
}
```

Mesa vs. Hoare monitors

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
```

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
```

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

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 (most textbooks):
    » Signaler gives 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

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 (most textbooks):
    - » Signaler gives 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
  - Mesa-style (most real operating systems):
    - » Signaler keeps lock and processor
    - » Waiter placed on ready queue with no special priority

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 (most textbooks):
    - » Signaler gives 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
  - Mesa-style (most real operating systems):
    - » Signaler keeps lock and processor
    - » Waiter placed on ready queue with no special priority
    - » Practically, need to check condition again after wait

Hoare Monitors: Semantics

- Hoare monitor semantics:
  - Assume thread *T1* is waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal
  - *T2* gives up monitor, *T2* blocks!
  - *T1* takes over monitor, runs
  - *T1* gives up monitor
  - *T2* takes over monitor, resumes
- Example

Hoare Monitors: Semantics

- Hoare monitor semantics:
  - Assume thread *T1* is waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal
  - *T2* gives up monitor, *T2* blocks!
  - *T1* takes over monitor, runs
  - *T1* gives up monitor
  - *T2* takes over monitor, resumes
- Example

```
fn1(…)
…
x.wait      // T1 blocks
```

Hoare Monitors: Semantics

- Hoare monitor semantics:
  - Assume thread *T1* is waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal
  - *T2* gives up monitor, *T2* blocks!
  - *T1* takes over monitor, runs
  - *T1* gives up monitor
  - *T2* takes over monitor, resumes

- Example

```
fn1(…)
…
x.wait      // T1 blocks   ——→   fn4(…)
                                 …
                                 x.signal   // T2 blocks
```

Hoare Monitors: Semantics

- Hoare monitor semantics:
  - Assume thread *T1* is waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal
  - *T2* gives up monitor, *T2* blocks!
  - *T1* takes over monitor, runs
  - *T1* gives up monitor
  - *T2* takes over monitor, resumes
- Example

```
fn1(…)
…
x.wait      // T1 blocks   ——→  fn4(…)
                                …
                          ←——   x.signal   // T2 blocks
// T1 resumes
Lock→release();
```

Hoare Monitors: Semantics

- Hoare monitor semantics:
  - Assume thread *T1* is waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal
  - *T2* gives up monitor, *T2* blocks!
  - *T1* takes over monitor, runs
  - *T1* gives up monitor
  - *T2* takes over monitor, resumes
- Example

```
fn1(…)
…
x.wait        // T1 blocks  ──────→  fn4(…)
                                     …
                     ←────────────   x.signal    // T2 blocks
// T1 resumes
Lock→release();   ──────────→
                                     T2 resumes
```

Hansen (Mesa) Monitors: Semantics

- Hansen monitor semantics:
  - Assume thread *T1* waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal; wake up T1
  - *T2* continues, finishes
  - When T1 get a chance to run, *T1* takes over monitor, runs
  - *T1* finishes, gives up monitor
- Example:

Hansen (Mesa) Monitors: Semantics

- Hansen monitor semantics:
  - Assume thread *T1* waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal; wake up T1
  - *T2* continues, finishes
  - When T1 get a chance to run, *T1* takes over monitor, runs
  - *T1* finishes, gives up monitor
- Example:

```
fn1(…)
…
x.wait      // T1 blocks
```

Hansen (Mesa) Monitors: Semantics

- Hansen monitor semantics:
  - Assume thread *T1* waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal; wake up T1
  - *T2* continues, finishes
  - When T1 get a chance to run, *T1* takes over monitor, runs
  - *T1* finishes, gives up monitor

- Example:

```
fn1(…)
…
x.wait      // T1 blocks   ───→   fn4(…)
                                  …
                                  x.signal   // T2 continues
                                  // T2 finishes
```

Hansen (Mesa) Monitors: Semantics

- Hansen monitor semantics:
  - Assume thread *T1* waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal; wake up T1
  - *T2* continues, finishes
  - When T1 get a chance to run, *T1* takes over monitor, runs
  - *T1* finishes, gives up monitor

- Example:

```
fn1(…)
…
x.wait      // T1 blocks    ──────→   fn4(…)
                                      …
                       ←──────────    x.signal   // T2 continues
                                      // T2 finishes

// T1 resumes
// T1 finishes
```

Tradeoff

Tradeoff

Hoare

- Claims:
  - Cleaner, good for proofs
  - When a condition variable is signaled, it does not change
  - Used in most textbooks
- …but
  - Inefficient implementation
  - Not modular – correctness depends on correct use and implementation of signal

Tradeoff

Hoare

- Claims:
  - Cleaner, good for proofs
  - When a condition variable is signaled, it does not change
  - Used in most textbooks
- ...but
  - Inefficient implementation
  - Not modular – correctness depends on correct use and implementation of signal

```
CokeMachine::Deposit(){
   lock→acquire();
   if (count == n) {
            notFull.wait(&lock); }
   Add coke to the machine;
   count++;
   notEmpty.signal();
   lock→release();
}
```

Tradeoff

Hoare

- Claims:
  - Cleaner, good for proofs
  - When a condition variable is signaled, it does not change
  - Used in most textbooks
- ...but
  - Inefficient implementation
  - Not modular – correctness depends on correct use and implementation of signal

```
CokeMachine::Deposit(){
    lock→acquire();
    if (count == n) {
            notFull.wait(&lock); }
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock→release();
}
```

Hansen

- Signal is only a hint that the condition may be true
  - Need to check condition again before proceeding
  - Can lead to synchronization bugs
- Used by most systems (e.g., Java)
- Benefits:
  - Efficient implementation
  - Condition guaranteed to be true once you are out of while !

Tradeoff

Hoare

- Claims:
  - Cleaner, good for proofs
  - When a condition variable is signaled, it does not change
  - Used in most textbooks
- ...but
  - Inefficient implementation
  - Not modular – correctness depends on correct use and implementation of signal

```
CokeMachine::Deposit(){
   lock→acquire();
   if (count == n) {
            notFull.wait(&lock); }
   Add coke to the machine;
   count++;
   notEmpty.signal();
   lock→release();
}
```

Hansen

- Signal is only a hint that the condition may be true
  - Need to check condition again before proceeding
  - Can lead to synchronization bugs
- Used by most systems (e.g., Java)
- Benefits:
  - Efficient implementation
  - Condition guaranteed to be true once you are out of while !

```
CokeMachine::Deposit(){
   lock→acquire();
   while (count == n) {
            notFull.wait(&lock); }
   Add coke to the machine;
   count++;
   notEmpty.signal();
   lock→release();
}
```

Problems with Monitors

Nested Monitor Calls

Problems with Monitors

Nested Monitor Calls

```
CokeMachine::Deposit(){
    lock→acquire();
    while (count == n) {
            notFull.wait(&lock); }
    truck->unload();
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock→release();
}
```

Problems with Monitors

Nested Monitor Calls

```
CokeMachine::Deposit(){
    lock→acquire();
    while (count == n) {
            notFull.wait(&lock); }
    truck->unload();
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock→release();
}
```

```
CokeTruck::Unload(){
    lock→acquire();
    while (soda.atDoor() != coke) {
            cokeAvailable.wait(&lock);}
    Unload soda closest to door;
    soda.pop();
    Signal availability for soda.atDoor();
    lock→release();
}
```

Problems with Monitors

Nested Monitor Calls

- What happens when one monitor calls into another?

```
CokeMachine::Deposit(){
    lock→acquire();
    while (count == n) {
            notFull.wait(&lock); }
    truck->unload();
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock→release();
}
```

```
CokeTruck::Unload(){
    lock→acquire();
    while (soda.atDoor() != coke) {
            cokeAvailable.wait(&lock);}
    Unload soda closest to door;
    soda.pop();
    Signal availability for soda.atDoor();
    lock→release();
}
```

Problems with Monitors

Nested Monitor Calls

- What happens when one monitor calls into another?
  - What happens to CokeMachine::lock if thread sleeps in CokeTruck::Unload?

```
CokeMachine::Deposit(){
    lock→acquire();
    while (count == n) {
            notFull.wait(&lock); }
    truck->unload();
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock→release();
}
```

```
CokeTruck::Unload(){
    lock→acquire();
    while (soda.atDoor() != coke) {
            cokeAvailable.wait(&lock);}
    Unload soda closest to door;
    soda.pop();
    Signal availability for soda.atDoor();
    lock→release();
}
```

Problems with Monitors

Nested Monitor Calls

- What happens when one monitor calls into another?
  - What happens to CokeMachine::lock if thread sleeps in CokeTruck::Unload?
  - What happens if truck unloader wants a coke?

```
CokeMachine::Deposit(){
    lock→acquire();
    while (count == n) {
            notFull.wait(&lock); }
    truck->unload();
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock→release();
}
```

```
CokeTruck::Unload(){
    lock→acquire();
    while (soda.atDoor() != coke) {
            cokeAvailable.wait(&lock);}
    Unload soda closest to door;
    soda.pop();
    Signal availability for soda.atDoor();
    lock→release();
}
```

More Monitor Headaches

The *priority inversion* problem

More Monitor Headaches

The *priority inversion* problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor *M.* P3 is the highest priority process, followed by P2 and P1.

More Monitor Headaches

The *priority inversion* problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor *M.* P3 is the highest priority process, followed by P2 and P1.
- 1. P1 enters M.

More Monitor Headaches

The *priority inversion* problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor *M.* P3 is the highest priority process, followed by P2 and P1.
- 1. P1 enters M.
- 2. P1 is preempted by P2.

More Monitor Headaches

The *priority inversion* problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor *M.* P3 is the highest priority process, followed by P2 and P1.
- 1. P1 enters M.
- 2. P1 is preempted by P2.
- 3. P2 is preempted by P3.

More Monitor Headaches

The *priority inversion* problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor *M.* P3 is the highest priority process, followed by P2 and P1.
- 1. P1 enters M.
- 2. P1 is preempted by P2.
- 3. P2 is preempted by P3.
- 4. P3 tries to enter the monitor, and waits for the lock.

More Monitor Headaches

The *priority inversion* problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor *M.* P3 is the highest priority process, followed by P2 and P1.
- 1. P1 enters M.
- 2. P1 is preempted by P2.
- 3. P2 is preempted by P3.
- 4. P3 tries to enter the monitor, and waits for the lock.
- 5. P2 runs again, preventing P3 from running, subverting the priority system.

More Monitor Headaches

The *priority inversion* problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor *M.* P3 is the highest priority process, followed by P2 and P1.
- 1. P1 enters M.
- 2. P1 is preempted by P2.
- 3. P2 is preempted by P3.
- 4. P3 tries to enter the monitor, and waits for the lock.
- 5. P2 runs again, preventing P3 from running, subverting the priority system.
- A simple way to avoid this situation is to associate with each monitor the priority of the highest priority process which ever enters that monitor.

Other Interesting Topics

- Exception handling
  - What if a process waiting in a monitor needs to time out?
- Naked notify
  - How do we synchronize with I/O devices that do not grab monitor locks, but can notify condition variables.
- Butler Lampson and David Redell, “Experience with Processes and Monitors in Mesa.”

Condition Variables

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 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

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

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!

Complete Monitor Example (with condition variable)

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

Complete Monitor Example (with condition variable)

- **Here is an (infinite) synchronized queue**

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

Complete Monitor Example (with condition variable)

- **Here is an (infinite) synchronized queue**

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

Complete Monitor Example (with condition 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
}
```

Complete Monitor Example (with condition 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);
}
```

Questions

Questions

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

```
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!
```

Questions

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

```
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?

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

Questions

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

```
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?**

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

Questions

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

```
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?

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

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

```
AR--;                       // No longer active
okToWrite.broadcast();      // Wake up one writer
```

Questions

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

```
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?

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

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

```
AR--;                     // No longer active
okToWrite.broadcast();    // Wake up one writer
```

- Finally, what if we use only one condition variable (call it "`okToContinue`") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Can we construct Monitors from Semaphores?

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex

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(); }
```

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(); }
```

  - Doesn't work: Wait() may sleep with lock held

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(); }
```

  - Doesn't work: Wait() may sleep with lock held
- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

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(); }
```

  - Doesn't work: Wait() may sleep with lock held
- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

  - No: Condition vars have no history, semaphores have history:

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(); }
```

  - Doesn't work: Wait() may sleep with lock held
- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

  - No: Condition vars have no history, semaphores have history:
    - » What if thread signals and no one is waiting? NO-OP

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(); }
```

  - Doesn't work: Wait() may sleep with lock held
- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

  - No: Condition vars have no history, semaphores have history:
    - » What if thread signals and no one is waiting? NO-OP
    - » What if thread later waits? Thread Waits

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(); }
```

  - Doesn't work: Wait() may sleep with lock held
- Does this work better?

```
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() { semaphore.V(); }
```

  - No: Condition vars have no history, semaphores have history:
    - » What if thread signals and no one is waiting? NO-OP
    - » What if thread later waits? Thread Waits
    - » What if thread V's and noone is waiting? Increment

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(); }
```

  - Doesn't work: Wait() may sleep with lock held
- Does this work better?

```
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() { semaphore.V(); }
```

  - No: Condition vars have no history, semaphores have history:
    - » What if thread signals and no one is waiting? NO-OP
    - » What if thread later waits? Thread Waits
    - » What if thread V's and noone is waiting? Increment
    - » What if thread later does P? Decrement and continue

Construction of Monitors from Semaphores (con't)

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

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?

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

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?

```
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

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?

```
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()

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?

```
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

Monitor Conclusion

Monitor Conclusion

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

Monitor Conclusion

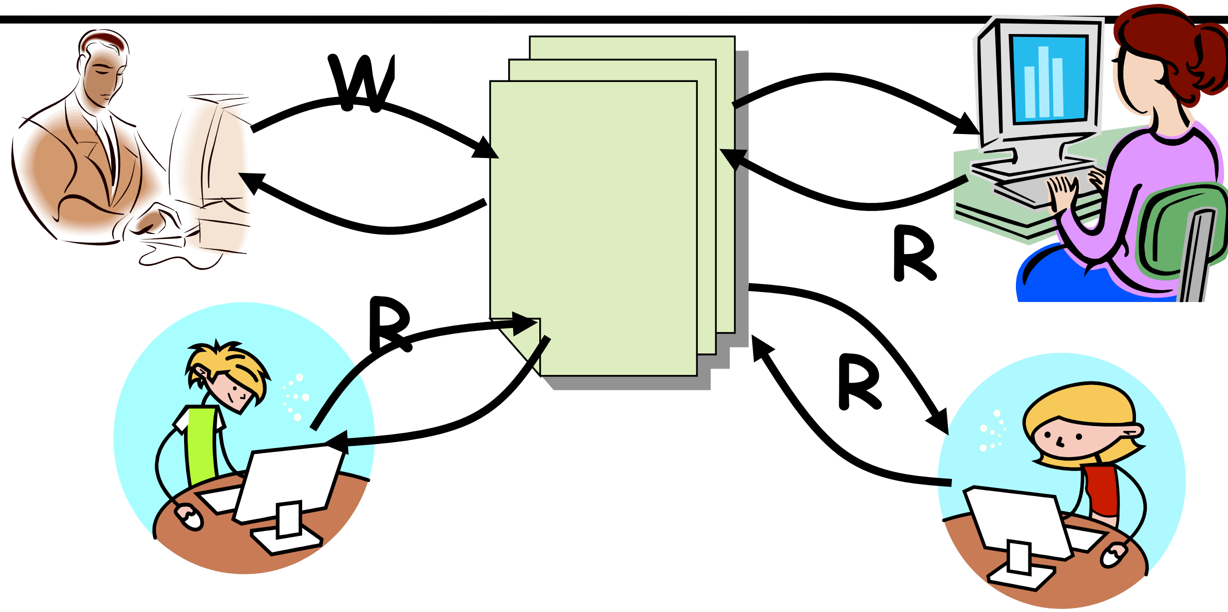
- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

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

do something so no need to wait

lock

condvar.signal();

unlock
```

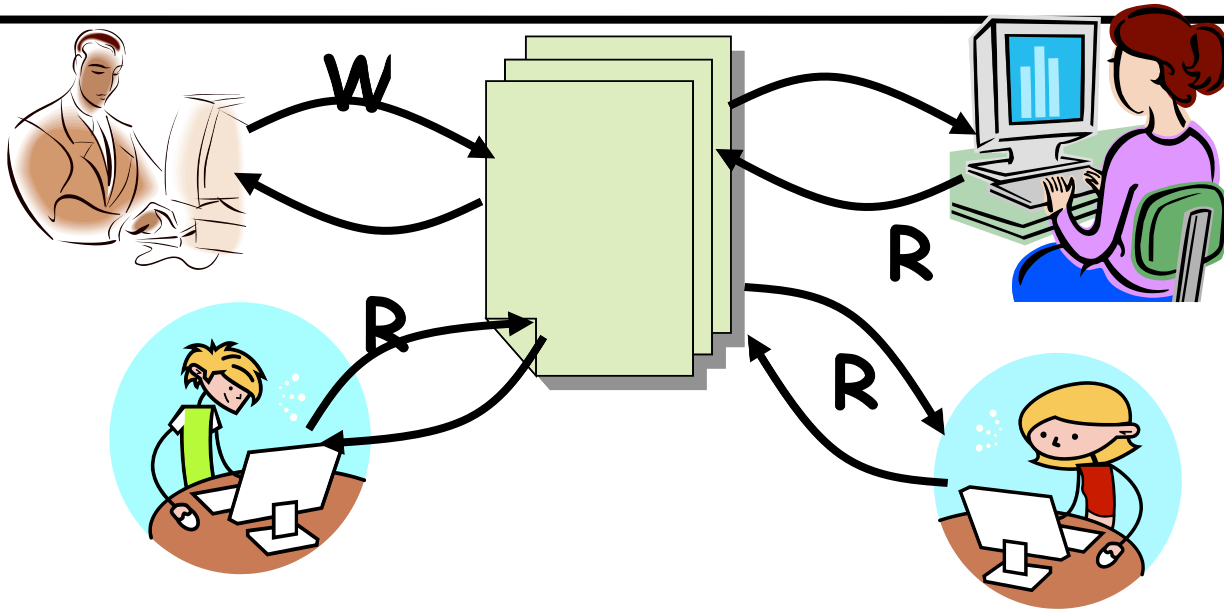
Check and/or update state variables
Wait if necessary

Check and/or update state variables

Readers/Writers Problem

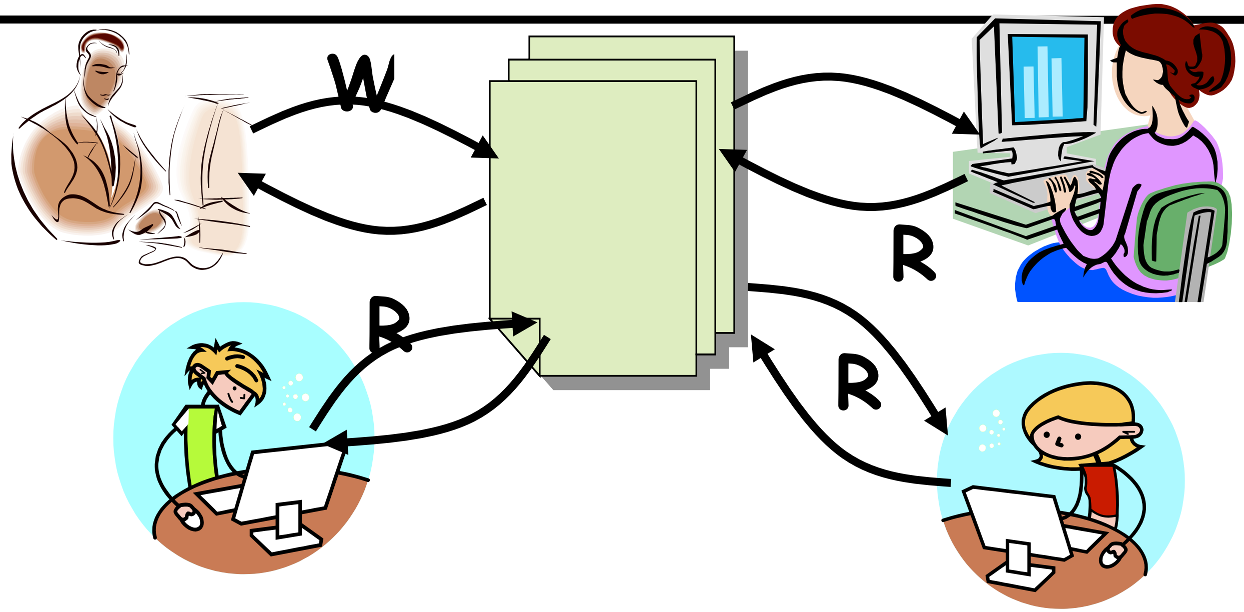


Readers/Writers Problem



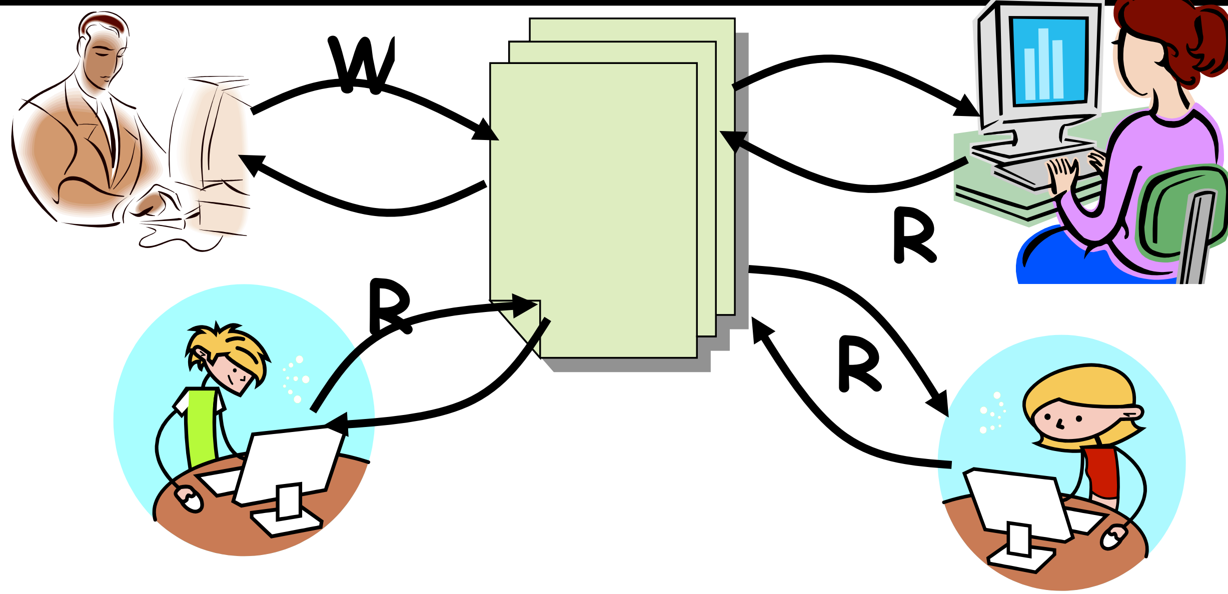
- **Motivation: Consider a shared database**

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database

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 Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:

Basic Readers/Writers Solution

- **Correctness Constraints:**
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- **Basic structure of a solution:**
  - Reader()

```
Wait until no writers
Access data base
Check out – wake up a waiting writer
```

Basic Readers/Writers Solution

- **Correctness Constraints:**
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- **Basic structure of a solution:**
  - Reader()
    ```
    Wait until no writers
    Access data base
    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
    ```

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - `Reader()`
    ```
    Wait until no writers
    Access data base
    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

Code for a Reader

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
```

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
  }
```

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();
```

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);
```

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 Reader

```
Reader() {
  // First check self into system
  lock.Acquire();

  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                        Writers exist
    okToRead.wai                 ep on cond var
    WR--;                        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();
}
```

Why Release the Lock here?

Code for a Writer

Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
```

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
  }
```

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();
```

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);
```

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();
}
```

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 s
  lock.Acquire();
  AW--;                          ctive
  if (WW > 0){                   ty to writers
    okToWrite.signal();          e writer
  } else if (WR > 0) {          Otherwise, wake reader
    okToRead.broadcast();  // Wake all readers
  }
  lock.Release();
}
```

Why broadcast() here instead of signal()?

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 r
  AccessDatabase(Read
  // Now, check out o
  lock.Acquire();
  AW--;                                 ctive
  if (WW > 0){                          ty to writers
     okToWrite.signal();                e writer
  } else if (WR > 0) {                  Otherwise, wake reader
     okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```

Why Give priority

Why broadcast() here instead of signal()?

Simulation of Readers/Writers solution

Simulation of Readers/Writers solution

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

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
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!
```

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
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!
```

- First, R1 comes along:

  AR = 1, WR = 0, AW = 0, WW = 0

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
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!
```

- First, R1 comes along:

  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:

  AR = 2, WR = 0, AW = 0, WW = 0

Simulation(2)

Simulation(2)

- **Next, W1 comes along:**

```
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++;
```

Simulation(2)

- **Next, W1 comes along:**

```
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++;
```

- **Can't start because of readers, so go to sleep:**

  AR = 2, WR = 0, AW = 0, WW = 1

Simulation(2)

- Next, W1 comes along:

```
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++;
```

- Can't start because of readers, so go to sleep:

  AR = 2, WR = 0, AW = 0, WW = 1

- Finally, R3 comes along:

  AR = 2, WR = 1, AW = 0, WW = 1

Simulation(2)

- Next, W1 comes along:

```
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++;
```

- Can't start because of readers, so go to sleep:

  AR = 2, WR = 0, AW = 0, WW = 1

- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1

Simulation(2)

- Next, W1 comes along:

```
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++;
```

- Can't start because of readers, so go to sleep:

  AR = 2, WR = 0, AW = 0, WW = 1

- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:

Simulation(2)

- Next, W1 comes along:

```
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++;
```

- Can't start because of readers, so go to sleep:

  AR = 2, WR = 0, AW = 0, WW = 1

- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:

```
if (AR == 0 && WW > 0)   // No other active readers
   okToWrite.signal();   // Wake up one writer
```

Simulation(3)

Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0

Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

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

Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

```
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!
```

  - Writer wakes up reader, so get:

    AR = 1, WR = 0, AW = 0, WW = 0

Simulation(3)

- When writer wakes up, get:

  AR = 0, WR = 1, AW = 1, WW = 0

- Then, when writer finishes:

```
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!
```

  - Writer wakes up reader, so get:

  AR = 1, WR = 0, AW = 0, WW = 0

- When writer completes, we are finished

C-Language Support for Synchronization

C-Language Support for Synchronization

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

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

C-Language Support for Synchronization

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

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

  - Watch out for `setjmp/longmp`!

C-Language Support for Synchronization

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

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

  - Watch out for `setjmp/longmp`!
  - Can cause a non-local jump out of procedure

C++ Language Support for Synchronization

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++ 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)
  - Consider:

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

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)
  - Consider:

```
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)

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  - Must catch exceptions, release lock, then re-throw the 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;
   …
}
```

Java Language Support for Synchronization

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization

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 Language Support for Synchronization (con't)

- **Java also has synchronized statements:**

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)

Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it

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();`

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`

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();
}
```

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

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

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()`

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()`
- Readers/Writers
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time

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()`
- Readers/Writers
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- Language support for synchronization:
  - Java provides synchronized keyword and one condition-variable per object (with `wait()` and `notify()`)