Monitors, Barriers, Readers/Writers

14 December 2025
Lecture 8

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Atomic Reads and Writes

Starvation

Race condition

Mutual exclusion

Critical section

Lock

- Spin lock
- In-Kernel lock

Busy waiting

Semaphores

Topics for Today

- Higher Level Synchronization Atoms
  - Monitors
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language

Concepts today

Why Monitors and Condition Variables?

Semaphores are a huge step up

- Try to do the bounded buffer with only loads and stores

Problem:

- Semaphores are dual purpose:
- They are used for both mutex and scheduling constraints

Example:

- That flipping P's in bounded buffer gives deadlock is not immediately obvious.
- How do you prove correctness to someone?

Cleaner idea:

- Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

What is a Monitor?

Monitor:

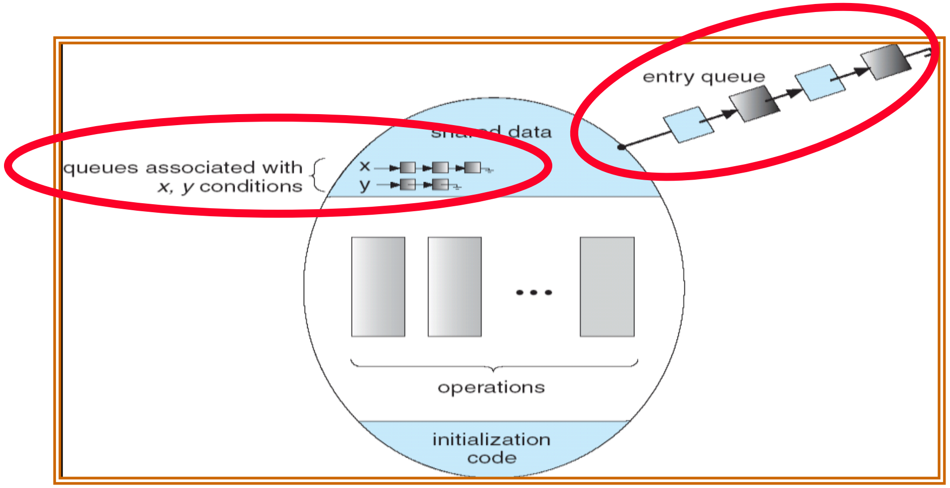
- A lock and zero or more condition variables for managing concurrent access to shared data

Some languages like Java provide this natively

Most others use actual locks and condition variables

Monitor with Condition Variables

- Lock: the lock provides mutual exclusion to shared data
  – Always acquire before accessing shared data structure
  – Always release after finishing with shared data
  – Lock initially free
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  – Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  – Contrast to semaphores: Can't wait inside critical section



Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;

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

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

- Not very interesting use of “Monitor”
  - It only uses a lock with no condition variables
  - Cannot put consumer to sleep if no work!

Condition Variables

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

- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Condition Variables

- Operations:

`Wait(&lock):`

- Atomically release lock and go to sleep. Reacquire 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)

- Here is an (infinite) synchronized queue

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

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

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

Mesa vs. Hoare monitors

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

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

- Why didn't we do this?

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

Mesa vs. Hoare scheduling

Hoare-style (textbooks):

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

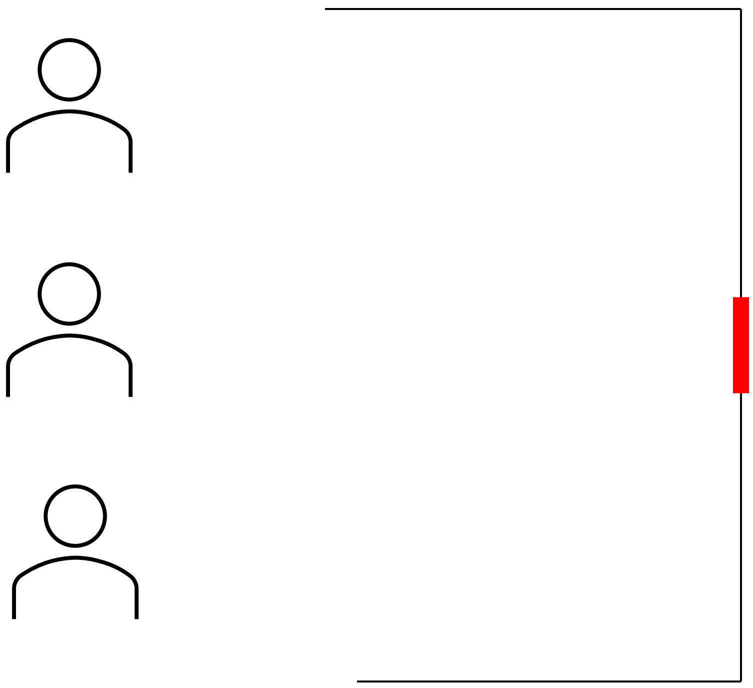
Mesa-style (most real OS):

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

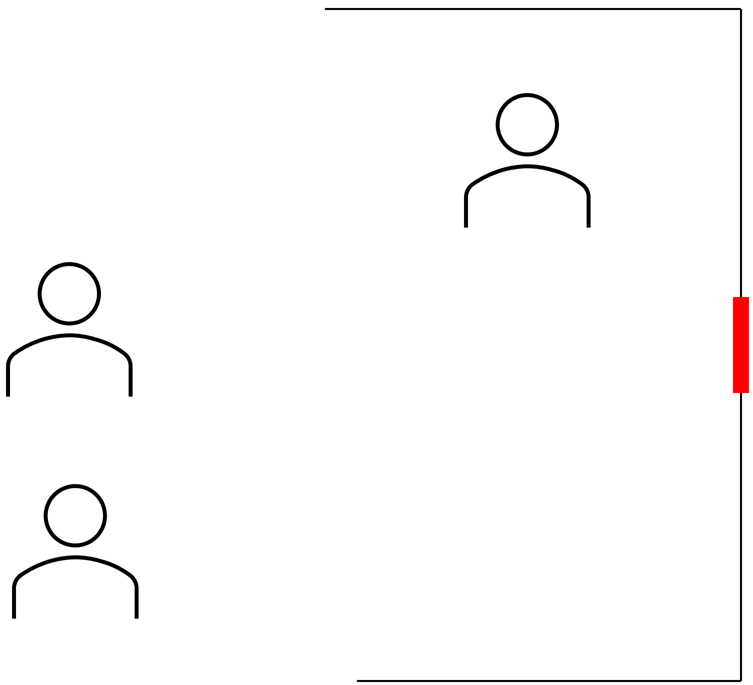
So Far

- Higher Level Synchronization Atoms
  - Monitors
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language

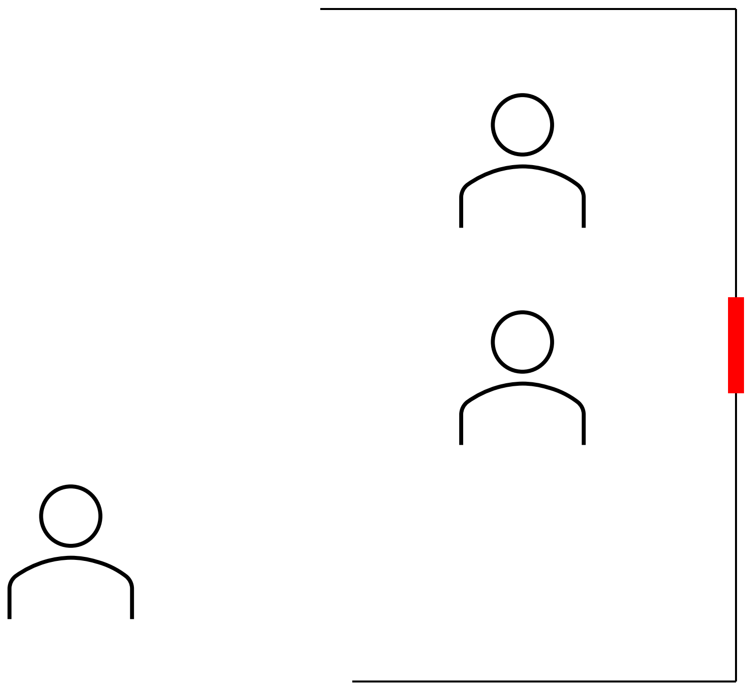
Barrier Synchronization - 3



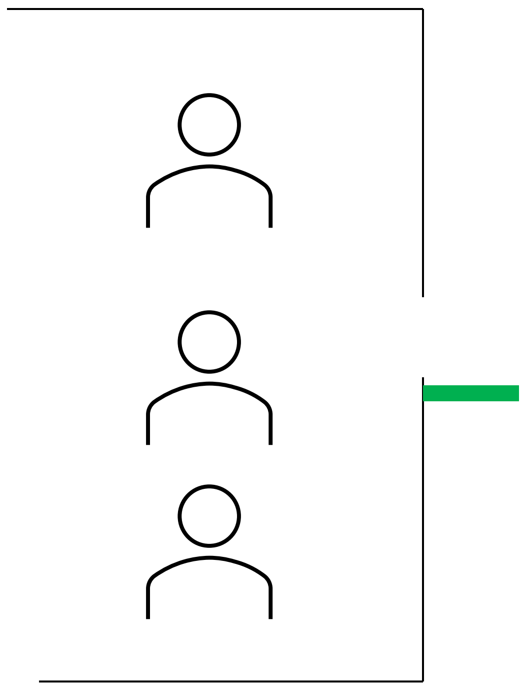
Barrier Synchronization - 3



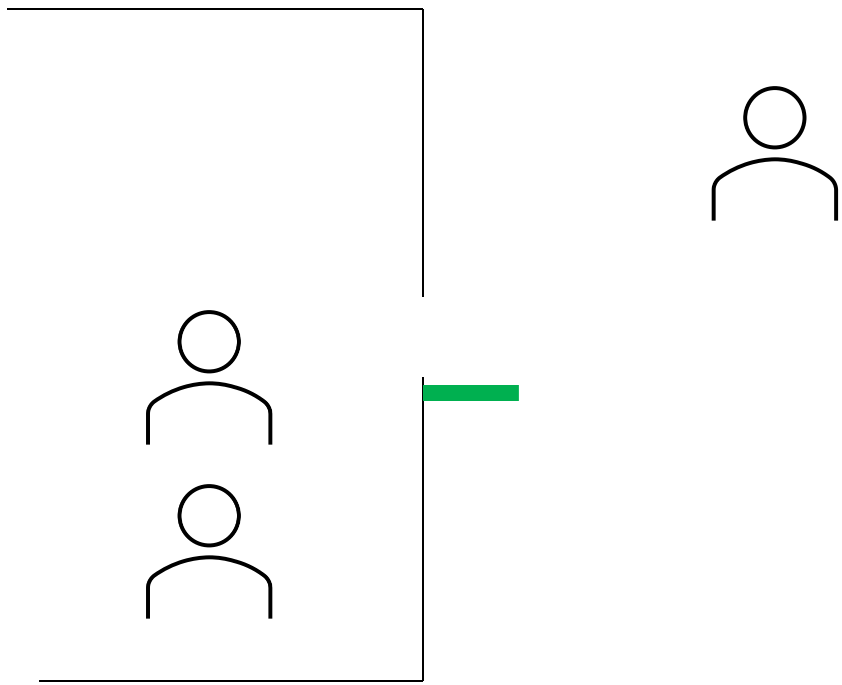
Barrier Synchronization - 3



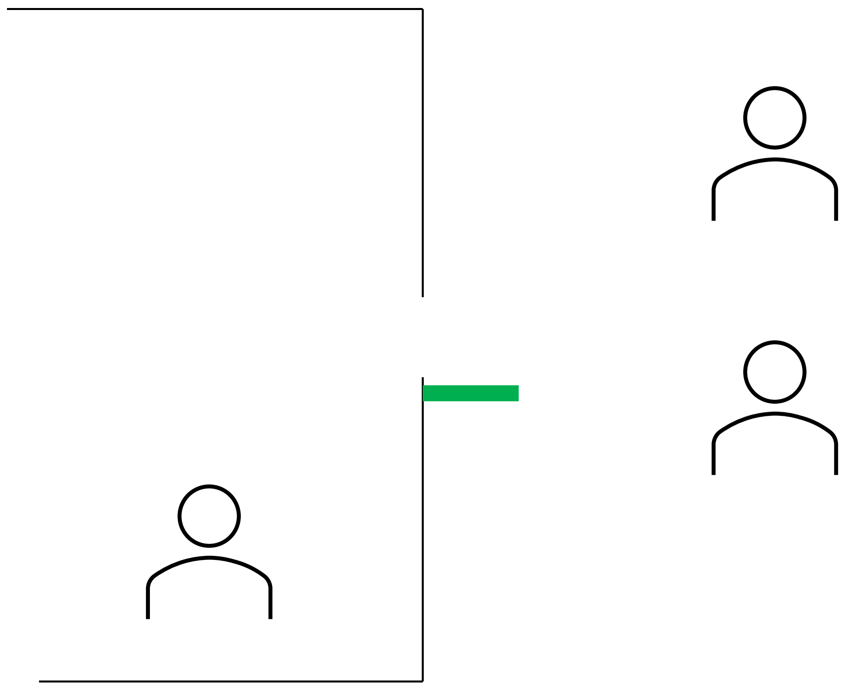
Barrier Synchronization - 3



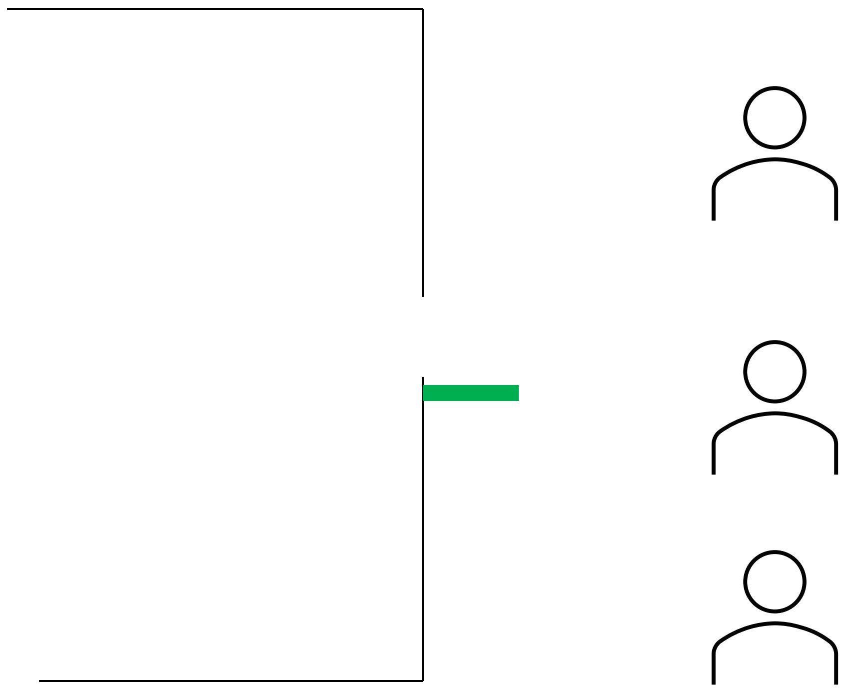
Barrier Synchronization - 3



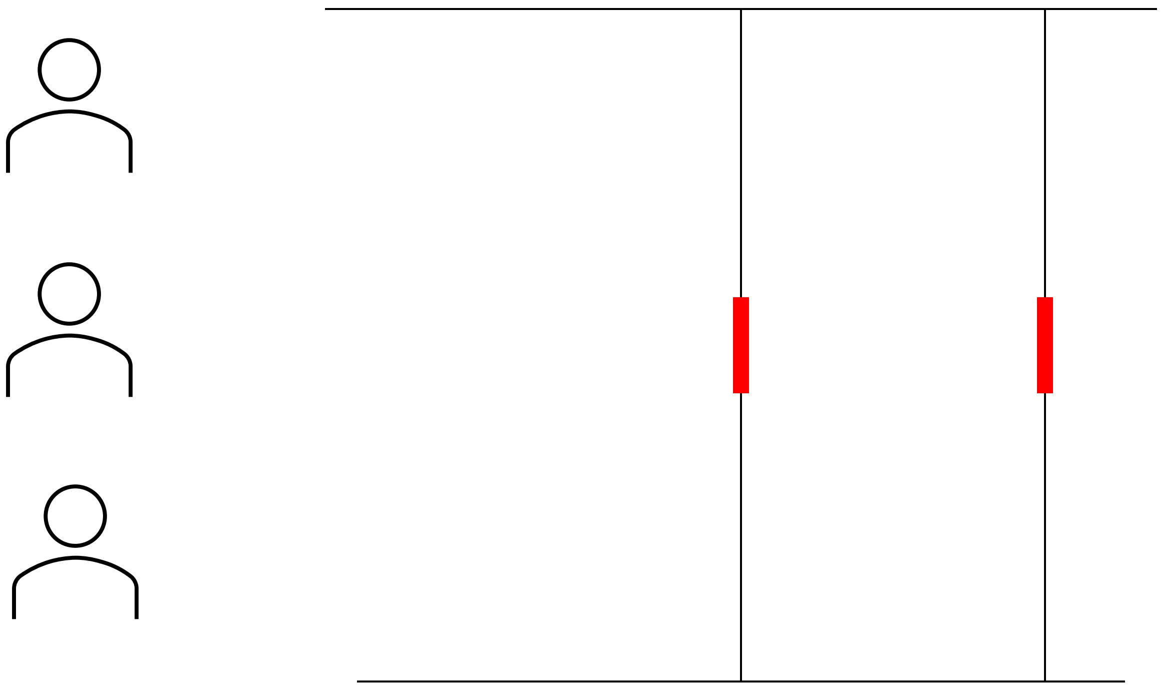
Barrier Synchronization - 3



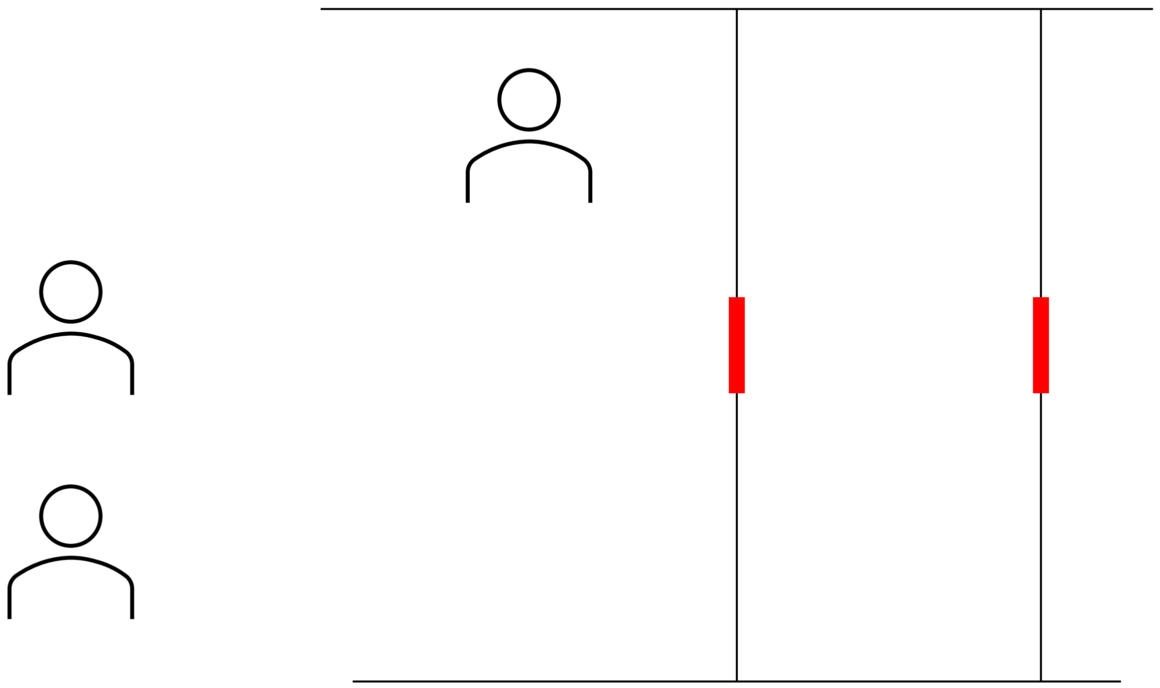
Barrier Synchronization - 3



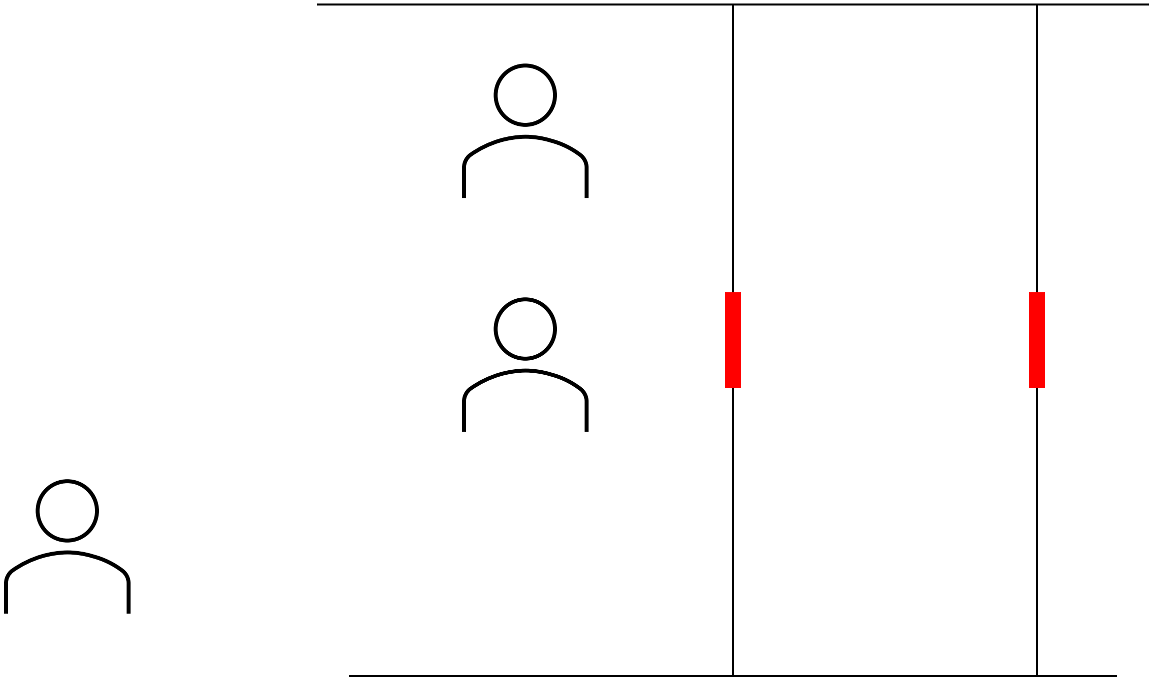
Reusable Barrier Synchronization - 3



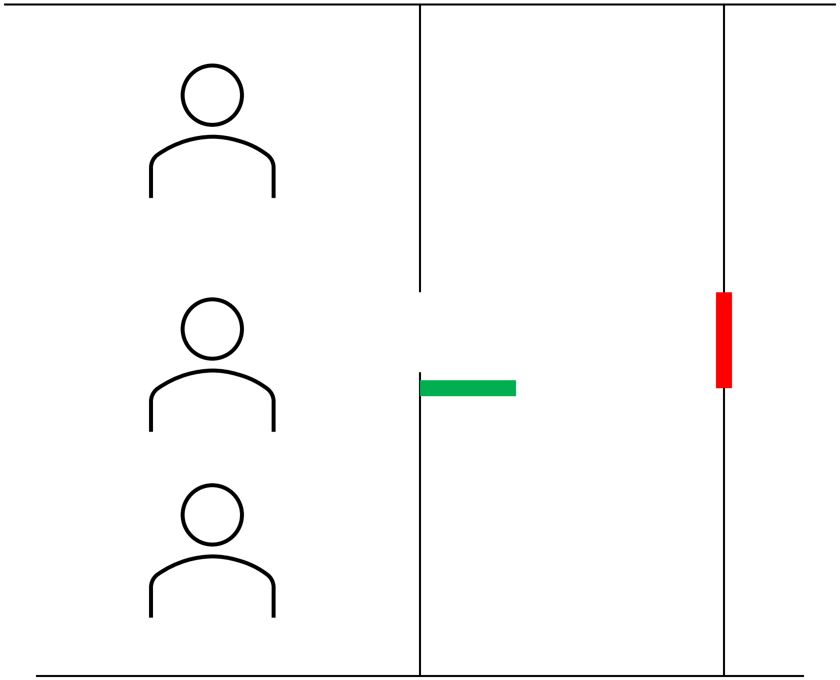
Reusable Barrier Synchronization - 3



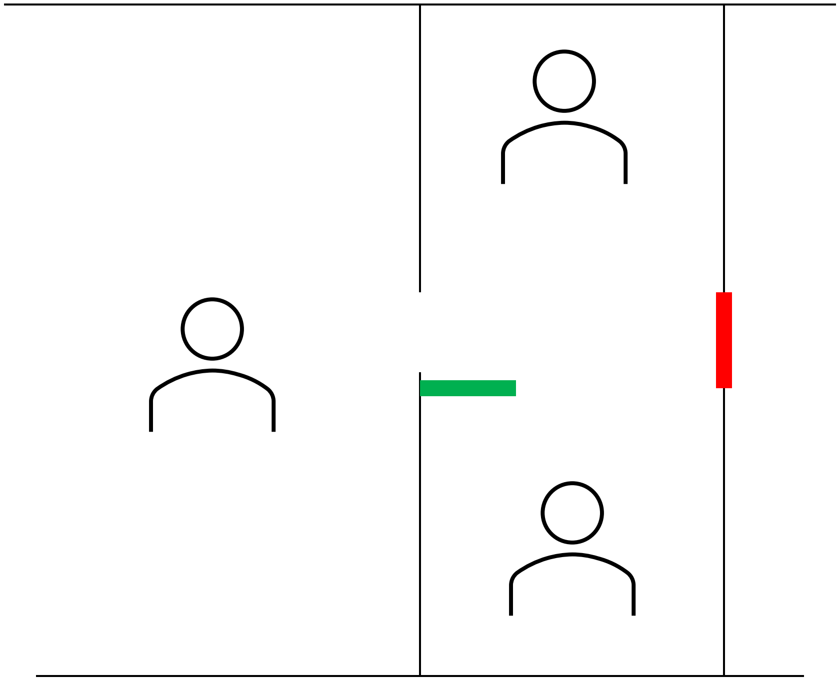
Reusable Barrier Synchronization - 3



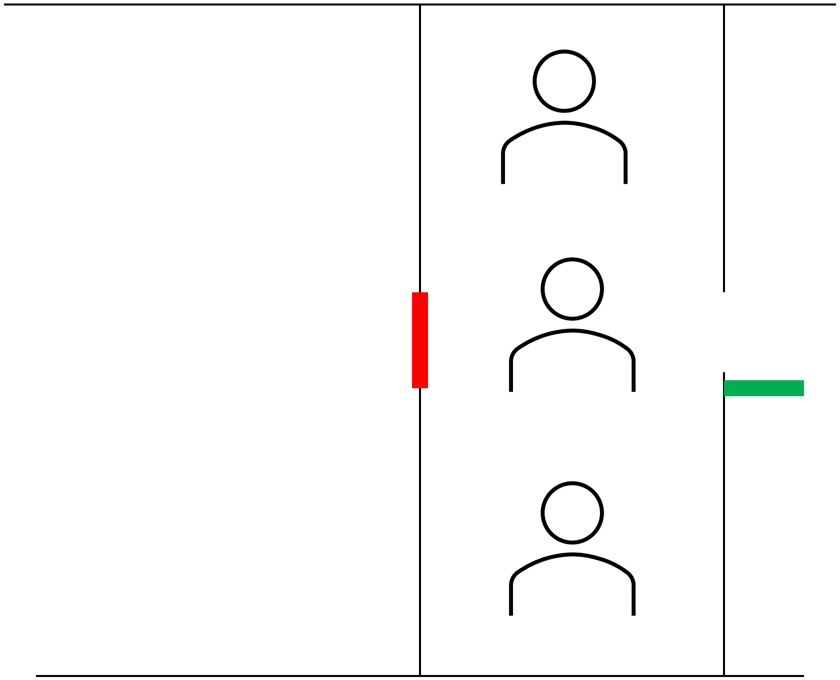
Reusable Barrier Synchronization - 3



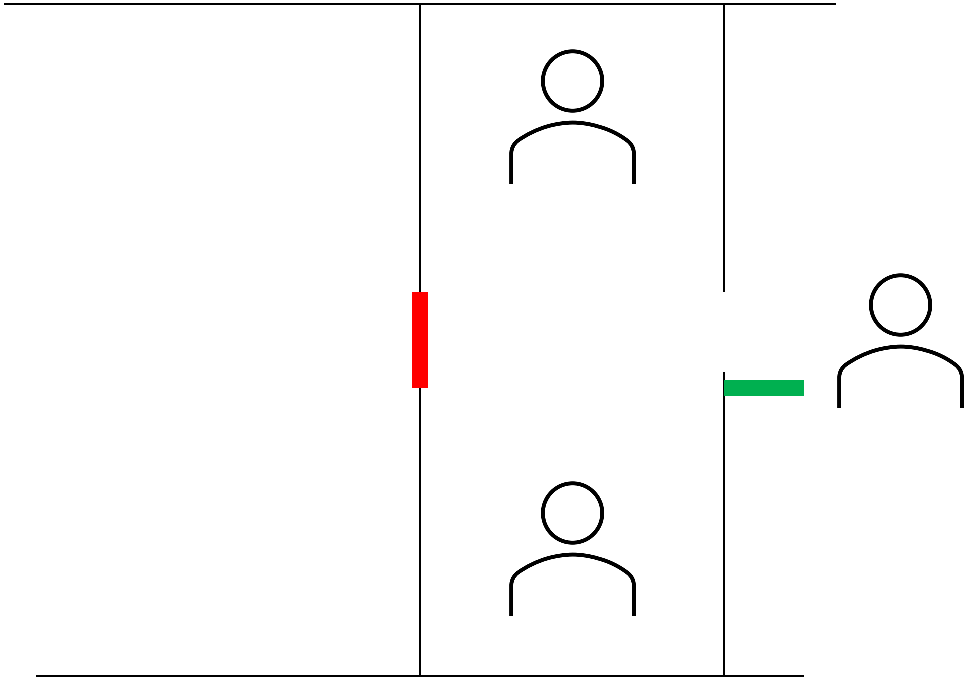
Reusable Barrier Synchronization - 3



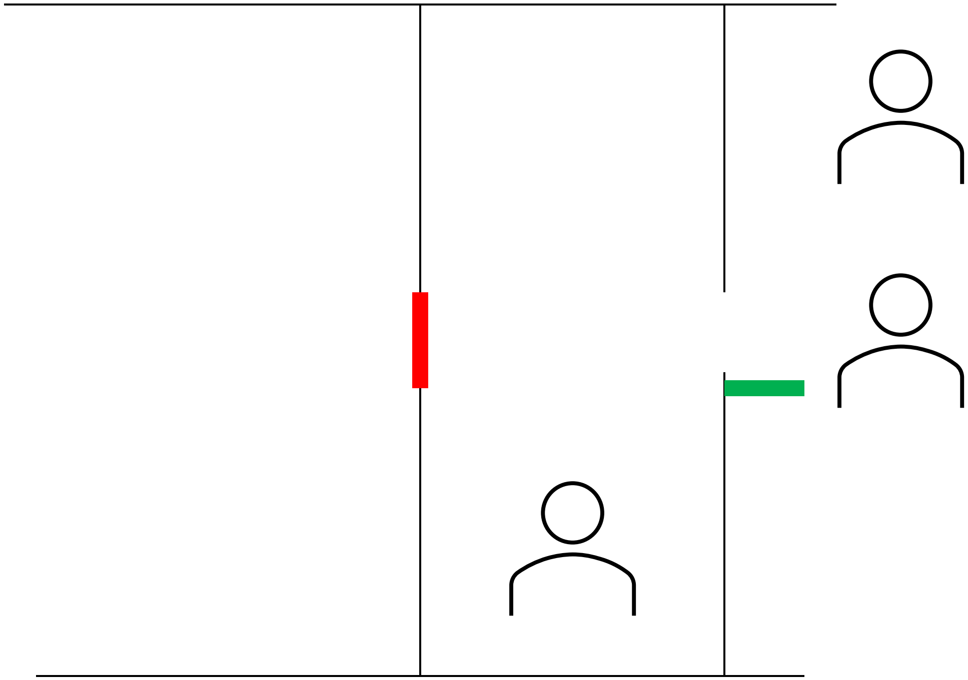
Reusable Barrier Synchronization - 3



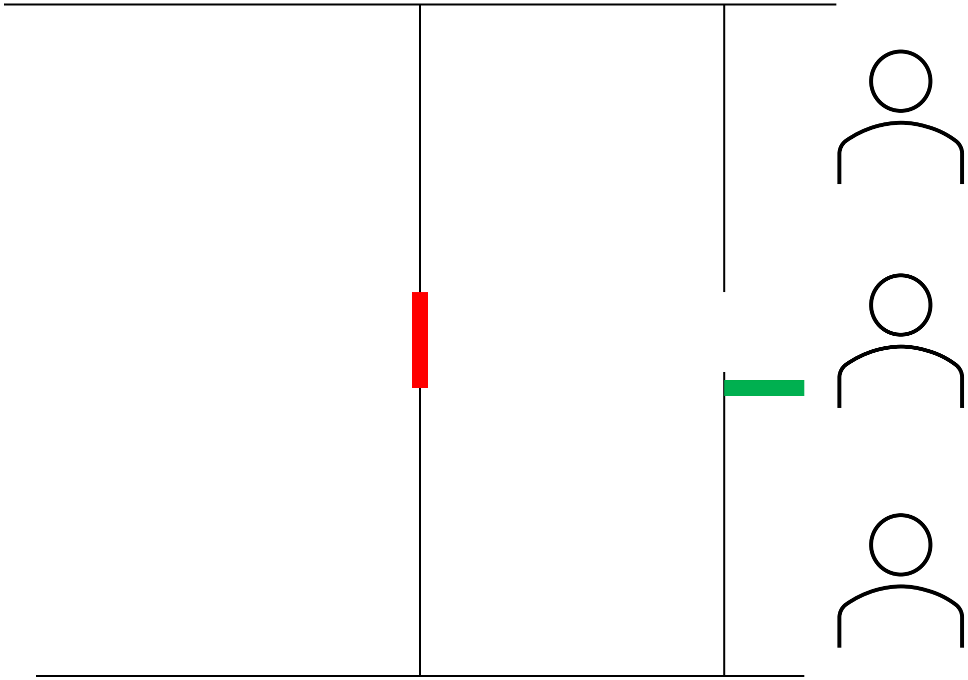
Reusable Barrier Synchronization - 3



Reusable Barrier Synchronization - 3



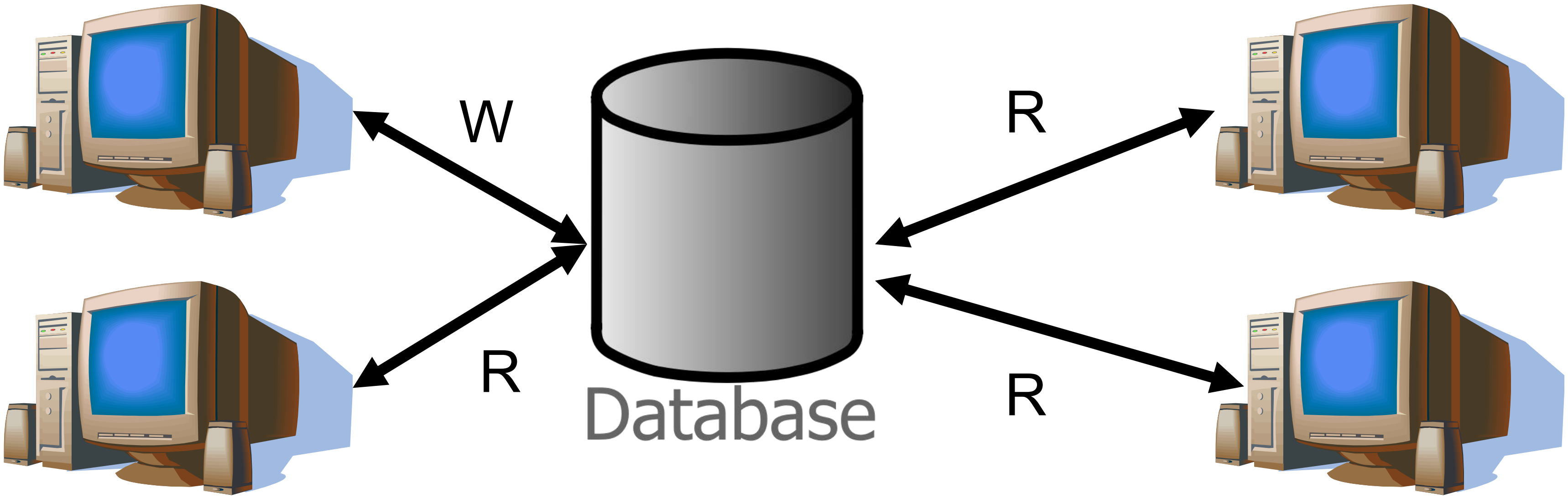
Reusable Barrier Synchronization - 3



So Far

- Higher Level Synchronization Atoms
  - Monitors
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language

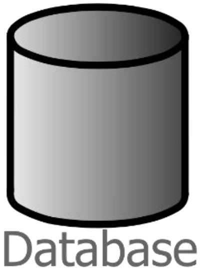
Extended example: Readers/Writers Problem



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?
  - Allow many readers at the same time
  - Only one writer at a time

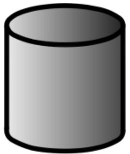
Basic Readers/Writers Solution



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

```
– Reader()
      Wait until no writers
      Access 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



- State variables (Protected by a lock called "lock"):
  - `int AR`: Number of active readers; initially = 0
  - `int WR`: Number of waiting readers; initially = 0
  - `int AW`: Number of active writers; initially = 0
  - `int WW`: Number of waiting writers; initially = 0
  - Condition `okToRead = NIL`
  - Condition `okToWrite = NIL`

Code for a Reader

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

Why? (arrow pointing to `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();
}
```

Why?

Why?

Simulation R/W Step 1

- 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`
- Second, `R2` comes along:
  `AR = 2, WR = 0, AW = 0, WW = 0`
- Now, readers make take a while to access database
  - Situation: Locks released
  - Only `AR` is non-zero

Simulation R/W Step 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 a writer:

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

Simulation R/W Step 3

- When the writer wakes up, get:

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

- Then, when writer finishes:

```
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
}
```

- Writer wakes up reader, so get:

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

- When reader completes, we are finished

Questions about R/W

- 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()`

Monitors Conclusion

- Monitors represent the logic of the program
  - `Wait` if necessary
  - `Signal` when you 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 or update state variables.
Wait if necessary

Check or update state variables.

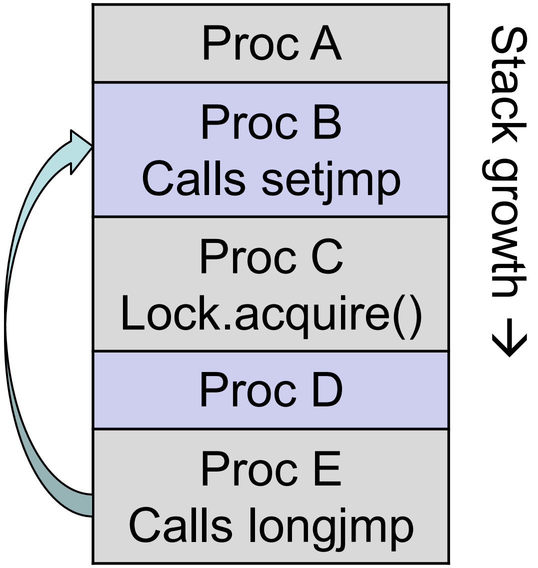
So Far

- Higher Level Synchronization Atoms
  - Monitors
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language

C-Language Support for Synchronization

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

C++ Language Support for Synchronization

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

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
  lock.acquire();
  try {
    …
    DoFoo();
    …
  } catch (…) {        // catch exception
    lock.release();    // release lock
    throw;             // re-throw the exception
  }
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Even Better: `auto_ptr<T>` facility. See C++ Spec.
    - Can deallocate/free lock regardless of exit method

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
   private int balance;
   // object constructor
   public Account (int initialBalance) {
      balance = initialBalance;
   }
   public synchronized int getBalance() {
      return balance;
   }
   public synchronized void deposit(int amount) {
      balance += amount;
   }
}
```

  – Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization

Java also has *synchronized* blocks:

```
int i, j;
void foo() {
   Object locker = new Object();
    synchronized (locker) {
      i += j;
    }
   }
```

- Since every Java object has <u>one</u> associated lock, the statement acquires and releases the object's lock on entry and exit of the block
- Problem is that the code here doesn't protect anything. Why?

Java Language Support for Synchronization

A better form of the code:

```
Object locker = new Object();
int i, j;
void foo() {
    synchronized (locker) {
      i += j;
    }
  }
```

- Now all threads will use the same lock and we'll get some mutual exclusion.

Java Language Support for Synchronization

- Works properly even with exceptions:

```
synchronized (locker) {
  …
  DoFoo();
  …
}
void DoFoo() {
  throw errException;
}
```

- Lock is released when the exception is thrown.

Java Language Support for Synchronization

- Every object also has <u>one</u> 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

- 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!

Conclusion

- Higher Level Synchronization Atoms
  - Monitors
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language