CS162
Operating Systems and Systems Programming
Lecture 8

Concurrency

Professor Natacha Crooks
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Higher-level Primitives than Locks

Goal of last couple of lectures:

- What is right abstraction for synchronizing threads that share memory?
- Want as high a level primitive as possible

Synchronization is a way of coordinating multiple concurrent activities that are using shared state

- This lecture and the next presents some ways of structuring sharing

Recall: Atomic Read-Write

- ```
  test&set (&address) {            /* most architectures */
      result = M[address];         // return result from "address" and
      M[address] = 1;              // set value at "address" to 1
      return result;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) { /* x86 (returns old value), 68000 */
      if (reg1 == M[address]) {    // If memory still == reg1,
          M[address] = reg2;       // then  put reg2 => memory
          return success;
      } else {                     // Otherwise do not change memory
          return failure;
      }
  }
  ```

Recall: futex - Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
          const struct timespec *timeout );
```

Interface to the kernel sleep() functionality!

- Let thread put themselves to sleep – conditionally!

futex is not exposed in libc; it is used within the implementation of pthreads

- Can be used to implement locks, semaphores, monitors, etc…

Recall: How to use a futex

```
bool maybe = false;
int mylock = 0; // Interface:
acquire(&mylock,&maybe_waiters);
                //
release(&mylock,&maybe_waiters);

acquire(int *thelock, bool *maybe) {
   while (test&set(thelock)) {
      // Sleep, since lock busy!
      *maybe = true;
      futex(thelock, FUTEX_WAIT, 1);

      // Make sure other sleepers not stuck
      *maybe = true;
   }
}
```

```
release(int*thelock, bool *maybe) {
   thelock = 0;
   if (*maybe) {
      *maybe = false;
      // Try to wake up someone
      futex(&thelock, FUTEX_WAKE, 1);
   }
}
```

This is syscall-free in the uncontended case

Temporarily falls back to syscalls if multiple waiters, or concurrent acquire/release

Semaphores

Semaphores are a type of generalized lock

First defined by Dijkstra in late 60s

Main synchronization primitive used in original UNIX

Semaphores

A Semaphore has a non-negative integer value and supports the following operations:

- Set value when you initialize
- `Down()` or `P()`: an atomic operation that waits for semaphore to become positive, then decrements it by 1
  - » Think of this as the wait() operation
- `Up()` or `V()`: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
  - » This of this as the signal() operation

Semaphores Like Integers Except…

Semaphores are like integers, except:

- No negative values
- Only operations allowed are P and V – can't read or write value, except initially
- Operations must be atomic
  - » Two P's together can't decrement value below zero
  - » Thread going to sleep in P won't miss wakeup from V – even if both happen at same time

Two Uses of Semaphores

Mutual Exclusion (initial value = 1)

Also called “Binary Semaphore” or “mutex”.

Can be used for mutual exclusion, just like a lock:

```
semaP(&mysem);
// Critical section goes here
semaV(&mysem);
```

Two Uses of Semaphores

Scheduling Constraints (initial value = 0)

Allow thread 1 to wait for a signal from thread 2

– thread 2 schedules thread 1 when a given event occurs

Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaP(&mysem);
}
ThreadFinish {
   semaV(&mysem);
}
```

Bounded Buffer: 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)

Bounded Buffer: Correctness constraints for solution

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

Let's drink coke!


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

  Producer(item) {
        semaP(&emptySlots);      // Wait until space




}
  Consumer() {
        semaP(&fullSlots);       // Check if there's a coke




}
```

Let’s drink coke!


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

  Producer(item) {
        semaP(&emptySlots);     // Wait until space
        Enqueue(item);



}
  Consumer() {
        semaP(&fullSlots);      // Check if there’s a coke
      item = Dequeue();



}
```

Let’s drink coke!


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

Producer(item) {
      semaP(&emptySlots);       // Wait until space
      Enqueue(item);
    semaV(&fullSlots);      // Tell consumers there is
                            // more coke


}
Consumer() {
      semaP(&fullSlots);        // Check if there’s a coke
    item = Dequeue();
    semaV(&emptySlots); // tell producer need more
    return item;
}
```

Let’s drink coke!

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

Producer(item) {
      semaP(&emptySlots);       // Wait until space
    semaP(&mutex);
      Enqueue(item);
    semaV(&mutex);
    semaV(&fullSlots);    // Tell consumers there is more coke
}
Consumer() {
      semaP(&fullSlots);        // Check if there’s a coke
    semaV(&mutex);
    item = Dequeue();
    semaV(&mutex);
    semaV(&emptySlots); // tell producer need more
    return item;
}
```

Let's drink coke!


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

Producer(item) {
      semaP(&emptySlots);       // Wait until space
      semaP(&mutex);        // Wait until machine free
      Enqueue(item);
      semaV(&mutex);
      semaV(&fullSlots);        // Tell consumers there
                            // more coke
}
Consumer() {
      semaP(&fullSlots);        // Check if there's a
      semaP(&mutex);        // Wait until machine free
      item = Dequeue();
      semaV(&mutex);
      semaV(&emptySlots);       // tell producer need more
      return item;
}
```

`fullSlots` signals coke

`emptySlots` signals space

Critical sections using mutex protect integrity of the queue

Discussion about Solution

Why asymmetry?

– Producer does: **`semaP(&emptyBuffer)`**, **`semaV(&fullBuffer)`**

– Consumer does: **`semaP(&fullBuffer)`**, **`semaV(&emptyBuffer)`**

Does order matter? What if we decrement mutex before full/emptyBuffer?

Semaphores are good but…

Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores or even with locks!

Problem is that semaphores are dual purpose:

- They are used for both mutex and scheduling constraints
- Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?

Monitors are better!

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

Monitor: a lock and zero or more condition variables for managing concurrent access to shared data

A monitor is a paradigm for concurrent programming

- Some languages like Java provide this natively
- Most others use actual locks and condition variables

Condition Variables

A queue of threads waiting for something (a condition) *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. 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!

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

Infinite Synchronized Buffer (with condition variable)

```
lock buf_lock;      // Initially unlocked
condition buf_CV; // Initially empty
queue queue;        // Actual queue!
```

Infinite Synchronized Buffer (with condition variable)

```
lock buf_lock;      // Initially unlocked
condition buf_CV; // Initially empty
queue queue;        // Actual queue!

Producer(item) {
      acquire(&buf_lock);      // Get Lock
      enqueue(&queue,item);    // Add item
      cond_signal(&buf_CV);    // Signal any
waiters
      release(&buf_lock);      // Release Lock
}
```

Infinite Synchronized Buffer (with condition variable)

```
lock buf_lock;     // Initially unlocked
condition buf_CV; // Initially empty
queue queue;       // Actual queue!

Producer(item) {
      acquire(&buf_lock);     // Get Lock
      enqueue(&queue,item);   // Add item
      cond_signal(&buf_CV);   // Signal any
waiters
      release(&buf_lock);     // Release Lock
}

Consumer() {
      acquire(&buf_lock);     // Get Lock
      if (isEmpty(&queue)) {
            cond_wait(&buf_CV, &buf_lock); // If
empty, sleep
      }
      item = dequeue(&queue); // Get next item
      release(&buf_lock);     // Release Lock
      return(item);
}
```

Infinite Synchronized Buffer (with condition variable)

```
lock buf_lock;     // Initially unlocked
condition buf_CV; // Initially empty
queue queue;       // Actual queue!

Producer(item) {
      acquire(&buf_lock);      // Get Lock
      enqueue(&queue,item);    // Add item
      cond_signal(&buf_CV);    // Signal any
waiters
      release(&buf_lock);      // Release Lock
}

Consumer() {
      acquire(&buf_lock);      // Get Lock
      while (isEmpty(&queue)) {
            cond_wait(&buf_CV, &buf_lock); // If
empty, sleep
      }
      item = dequeue(&queue); // Get next item
      release(&buf_lock);      // Release Lock
      return(item);
}
```

Mesa vs. Hoare monitors

Need to be careful about precise definition of signal and wait.

```
while (isEmpty(&queue)) {
      cond_wait(&buf_CV,&buf_lock); // If nothing, sleep
}
item = dequeue(&queue); // Get next item
```

Why didn't we do this?

```
if (isEmpty(&queue)) {
      cond_wait(&buf_CV,&buf_lock); // If nothing, sleep
}
item = dequeue(&queue); // Get next item
```

Answer: depends on the type of scheduling

- Mesa-style: Named after Xerox-Park Mesa Operating System
  - Most OSes use Mesa Scheduling!
- Hoare-style: Named after British logician Tony Hoare

Hoare monitors

Signaler gives up lock, CPU to waiter; waiter runs immediately

Then, Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again

```
…
acquire(&buf_lock);
…
cond_signal(&buf_CV);
…
release(&buf_lock);
```

Lock, CPU

Lock, CPU

```
acquire(&buf_lock);
…
if (isEmpty(&queue)) {
  cond_wait(&buf_CV,&buf_lock);
}
…
release(&buf_lock);
```

At first glance, this seems like good semantics

Waiter gets to run immediately, condition is still correct!

Mesa monitors

Signaler keeps lock and processor
Waiter placed on ready queue with no special priority

```
…
acquire(&buf_lock)
…
cond_signal(&buf_CV);
…
release(&buf_lock));
```

Put waiting thread on ready queue

schedule thread (sometime later!)

```
acquire(&buf_lock);
…
while (isEmpty(&queue)) {
  cond_wait(&buf_CV,&buf_lock)
;
}
…
lock.Release();
```

Practically, need to check condition again after wait

– By the time the waiter gets scheduled, condition may be false again

– – so, just check again with the "while" loop

Bounded Buffer – Attempt 4

```
lock buf_lock = <initially unlocked>
condition isNotEmpty = <initially empty>
condition isNotFull = <initially empty>
```

Bounded Buffer – Attempt 4

```
lock buf_lock = <initially unlocked>
condition isNotEmpty= <initially empty>
condition isNotFull = <initially empty>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) { cond_wait(&isNotFull,
&buf_lock); }
  enqueue(item);
  cond_signal(&isNotEmpty);
  release(&buf_lock);
}

Consumer() {
  acquire(buf_lock);
  while (buffer empty) { cond_wait(&isNotEmpty,
&buf_lock); }
  item = dequeue();
  cond_signal(&isNotFull);
  release(buf_lock);
  return item
}
```

Again: Why the `while` Loop?

MESA semantics

For most operating systems, when a thread is woken up by signal(), it is simply put on the ready queue

It may or may not reacquire the lock immediately!

- Another thread could be scheduled first and "sneak in" to empty the queue
- Need a loop to re-check condition on wakeup

Is this busy waiting?

Basic Structure of *Mesa* Monitor Program

Monitors represent the synchronization logic of the program

- Wait if necessary
- Signal when change something so any waiting threads can proceed

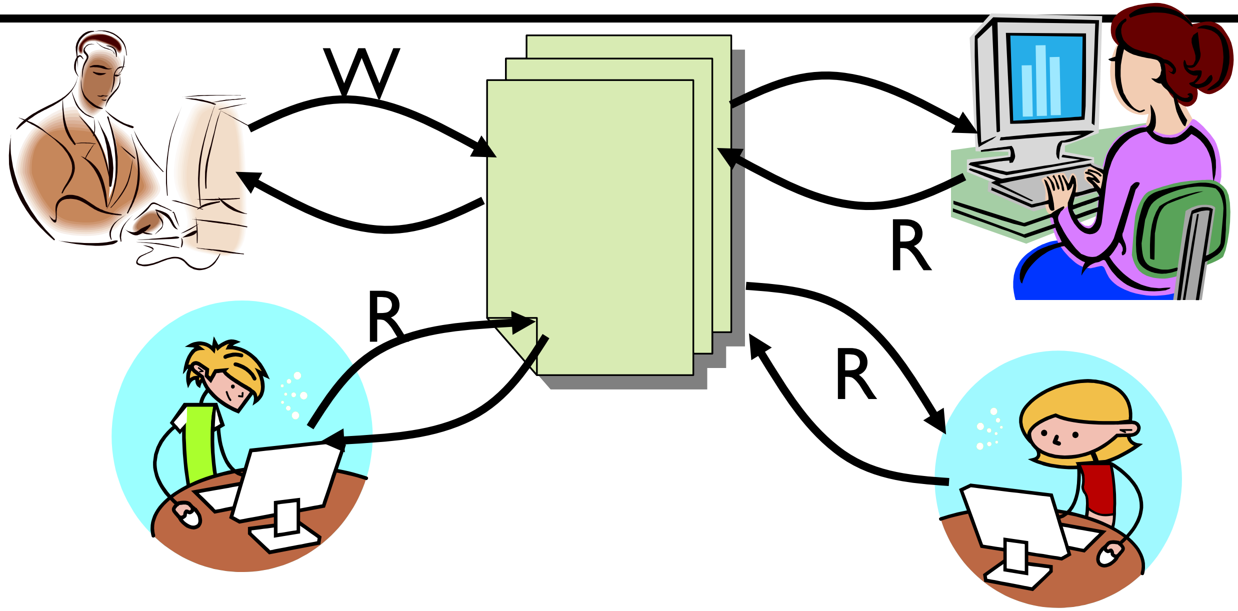
```
lock
while (need to wait) {
    condvar.wait();
}
unlock
```

Check and/or update state variables
Wait if necessary

```
do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables

Readers/Writers Problem



Motivation: Consider a shared database

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

Basic Readers/Writers Solution

Correctness Constraints:

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

Basic structure of a solution:

- **`Reader()`**
```
    Wait until no writers
    Access 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
      acquire(&lock);
      while ((AW + WW) > 0) { // Is it safe to read?
            WR++; // No. Writers exist
            cond_wait(&okToRead,&lock);// Sleep on cond
var
            WR--; // No longer waiting
      }
      AR++;       // Now we are active!
      release(&lock);
      // Perform actual read-only access
      AccessDatabase(ReadOnly);
      // Now, check out of system
      acquire(&lock);
      AR--;       // No longer active
      if (AR == 0 && WW > 0)  // No other active readers
            cond_signal(&okToWrite);// Wake up one writer
      release(&lock);
}
```

Code for a Writer

```
Writer() {
      // First check self into system
      acquire(&lock);
      while ((AW + AR) > 0) { // Is it safe to write?
            WW++; // No. Active users exist
            cond_wait(&okToWrite,&lock);  // Sleep on
cond var
            WW--; // No longer waiting
      }
      AW++;       // Now we are active!
      release(&lock);
      // Perform actual read/write access
      AccessDatabase(ReadWrite);
      // Now, check out of system
      acquire(&lock);
      AW--;       // No longer active
      if (WW > 0){// Give priority to writers
            cond_signal(&okToWrite);// Wake up one writer
      } else if (WR > 0) {    // Otherwise, wake reader
            cond_broadcast(&okToRead); // Wake all
readers
      }
      release(&lock);
}
```

Simulation of Readers/Writers Solution

Use an example to simulate the solution

Consider the following sequence of operators:

– R1, R2, W1, R3

Initially: AR = 0, WR = 0, AW = 0, WW = 0

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
                cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
                cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 accessing dbase (no other threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
}
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)

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

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

      AccessDBase(ReadOnly);

      acquire(&lock);
      AR--;
      if (AR == 0 && WW > 0)
            cond_signal(&okToWrite);
      release(&lock);
  }
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 and R2 accessing dbase

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
```

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

W1 comes along (R1 and R2 are still accessing dbase)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 comes along (R1 and R2 accessing dbase, W1 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 comes along (R1 and R2 accessing dbase, W1 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 comes along (R1 and R2 accessing dbase, W1 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 comes along (R1, R2 accessing dbase, W1 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 and R2 accessing dbase, W1 and R3 waiting

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        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 and R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 finishes (W1 and R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 finishes (W1, R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 finishes (W1, R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R1 signals a writer (W1 and R3 waiting)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

W1 gets signal (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

W1 gets signal (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

W1 gets signal (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

W1 accessing dbase (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
                cond_signal(&okToWrite);
        } else if (WR > 0) {
                cond_broadcast(&okToRead);
        }
        release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 finishes (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

W1 finishes (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

W1 finishes (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 signaling readers (R3 still waiting)

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

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

        AccessDBase(ReadWrite);

        acquire(&lock);
        AW--;
        if (WW > 0){
              cond_signal(&okToWrite);
        } else if (WR > 0) {
              cond_broadcast(&okToRead);
        }
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 gets signal (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 gets signal (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 accessing dbase (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 finishes (no waiting threads)

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

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

        AccessDBase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Simulation of Readers/Writers Solution

R3 finishes (no waiting threads)

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

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

        AccessDbase(ReadOnly);

        acquire(&lock);
        AR--;
        if (AR == 0 && WW > 0)
              cond_signal(&okToWrite);
        release(&lock);
  }
```

Questions

Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) { // Is it safe to read?
            WR++; // No. Writers exist
            cond_wait(&okToRead,&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
         cond_signal(&okToWrite);// Wake up one
writer
```

Questions

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

```
AR--; // No longer active
cond_broadcast(&okToWrite); // Wake up sleepers
```

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

- Both readers and writers sleep on this variable
- Must use broadcast() instead of signal()

Code for a Reader

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

Code for a Writer

```
Writer() {
      // First check self into system
      acquire(&lock);
      while ((AW + AR) > 0) { // Is it safe to write?
            WW++; // No. Active users exist
            cond_wait(&okToWrite,&lock);  // Sleep on
cond var
            WW--; // No longer waiting
      }
      AW++;       // Now we are active!
      release(&lock);
      // Perform actual read/write access
      AccessDatabase(ReadWrite);
      // Now, check out of system
      acquire(&lock);
      AW--;       // No longer active
      if (WW > 0){// Give priority to writers
            cond_signal(&okToWrite);// Wake up one writer
      } else if (WR > 0) {    // Otherwise, wake reader
            cond_broadcast(&okToRead); // Wake all
readers
      }
      release(&lock);
}
```

Mesa Monitor Conclusion

Monitors represent the synchronization logic of the program

- Wait if necessary
- Signal when change something so any waiting threads can proceed

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

Check and/or update state variables
Wait if necessary

```
do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables

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() {
            acquire(&lock);
            …
            if (exception) {
                  release(&lock);
                  return errReturnCode;
            }
            …
            release(&lock);
            return OK;
}
```

Concurrency and Synchronization in C

Harder with more locks

```
void Rtn() {
  lock1.acquire();
  …
  if (error) {
    lock1.release();
    return;
  }
  …
  lock2.acquire();
  …
  if (error) {
    lock2.release()
    lock1.release();
    return;
  }
  …
  lock2.release();
  lock1.release();
}
```

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)

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

- Notice that an exception in DoFoo() will exit without releasing the lock!

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

Must catch all exceptions in critical sections

– Catch exceptions, release lock, and re-throw exception:

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

Much better: C++ Lock Guards

```
#include <mutex>
int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
  std::lock_guard<std::mutex> lock(global_mutex);
  …
  global_i++;
  // Mutex released when 'lock' goes out of scope
}
```

Python `with` Keyword

More versatile than we show here (can be used to close files, database connections, etc.)

```
lock = threading.Lock()
…
with lock: # Automatically calls acquire()
  some_var += 1
  …
# release() called however we leave block
```

Java synchronized Keyword

Every Java object has an associated lock:

- Lock is acquired on entry and released on exit from a synchronized method
- Lock is properly released if exception occurs inside a synchronized method
  - Mutex execution of synchronized methods (beware deadlock)

```
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;
      }
}
```

Java Support for Monitors

Along with a lock, every object has a single condition variable associated with it

To wait inside a synchronized method:

- `void wait();`
- `void wait(long timeout);`

To signal while in a synchronized method:

- `void notify();`
- `void notifyAll();`

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

Implement various higher-level synchronization primitives using atomic operations