

Operating Systems (Honor Track)

Synchronization 4: Monitors and Readers/Writers

Xin Jin

Spring 2022

Recall: Too Much Milk Solution #3

- Here is a possible two-note solution:

<u>Thread A</u>	<u>Thread B</u>
leave note A;	leave note B;
while (note B) { \\X	if (noNote A) { \\Y
do nothing;	if (noMilk) {
}	buy milk;
if (noMilk) {	}
buy milk;	}
}	remove note B;
remove note A;	

- Does this work? **Yes**. Both can guarantee that:
 - It is safe to buy, or
 - Other will buy, ok to quit
- At X:
 - If no note B, safe for A to buy,
 - Otherwise wait to find out what will happen
- At Y:
 - If no note A, safe for B to buy
 - Otherwise, A is either buying or waiting for B to quit

Recall: Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
 - Recall: dispatcher gets control in two ways.
 - » Internal: Thread does something to relinquish the CPU
 - » External: Interrupts cause dispatcher to take CPU
 - On a uniprocessor, can avoid context-switching by:
 - » Avoiding internal events
 - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }  
LockRelease { enable Ints; }
```
- Problems with this approach:
 - **Can't let user do this!** Consider following:

```
LockAcquire();  
While(TRUE) {;
```
 - Real-Time system—no guarantees on timing!
 - » Critical Sections might be arbitrarily long
 - What happens with I/O or other important events?
 - » “Reactor about to meltdown. Help?”



Recall: Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```



```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

```
Release() {  
    disable interrupts;  
    if (anyone on wait queue) {  
        take thread off wait queue;  
        place on ready queue;  
    } else {  
        value = FREE;  
    }  
    enable interrupts;  
}
```

Recall: Implementing Locks with test&set

- Another flawed, but simple solution:

```
int value = 0; // Free
Acquire() {
    while (test&set(value)); // while busy
}
Release() {
    value = 0;
}
```

- Simple explanation:
 - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
 - If lock is busy, test&set reads 1 and sets value=1 (no change) It returns 1, so while loop continues.
 - When we set value = 0, someone else can get lock.
- **Busy-Waiting**: thread consumes cycles while waiting
 - For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network BW)

Recall: Better Locks using test&set

- Can we build test&set locks without busy-waiting?
 - Can't entirely, but can minimize!
 - Idea: only busy-wait to atomically check lock value

```
int guard = 0;  
int value = FREE;
```



```
Acquire() {  
    // Short busy-wait time  
    while (test&set(guard));  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep() & guard = 0;  
    } else {  
        value = BUSY;  
        guard = 0;  
    }  
}
```

```
Release() {  
    // Short busy-wait time  
    while (test&set(guard));  
    if anyone on wait queue {  
        take thread off wait queue  
        Place on ready queue;  
    } else {  
        value = FREE;  
    }  
    guard = 0;  
}
```

- Note: sleep has to be sure to reset the guard variable
 - Why can't we do it just before or just after the sleep?

Recall: Semaphores are good but...Monitors are better!

- 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?
- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: **Monitor**: a **lock** and zero or more **condition variables** for managing concurrent access to shared data
 - Some languages like Java provide this natively
 - Most others use actual locks and condition variables
- A “Monitor” is a paradigm for concurrent programming!
 - Some languages support monitors explicitly

Recall: Condition Variables

- How do we change the `consumer()` 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!

Recall: Synchronized Buffer (with condition variable)

- Here is an (infinite) synchronized queue:

```
lock buf_lock; // Initially unlocked
condition buf_CV; // Initially empty
queue 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. Consider a piece of our dequeue code:

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

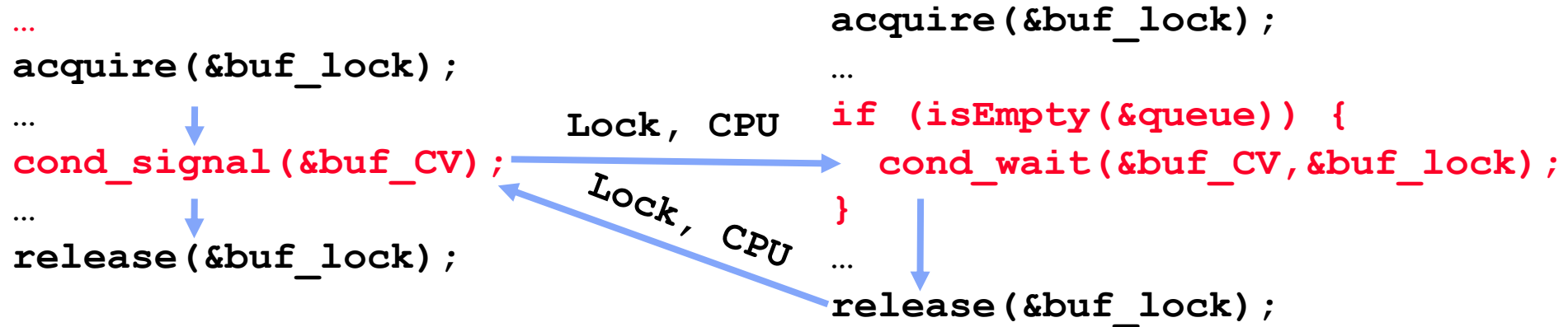
- Why didn't we do this?

```
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

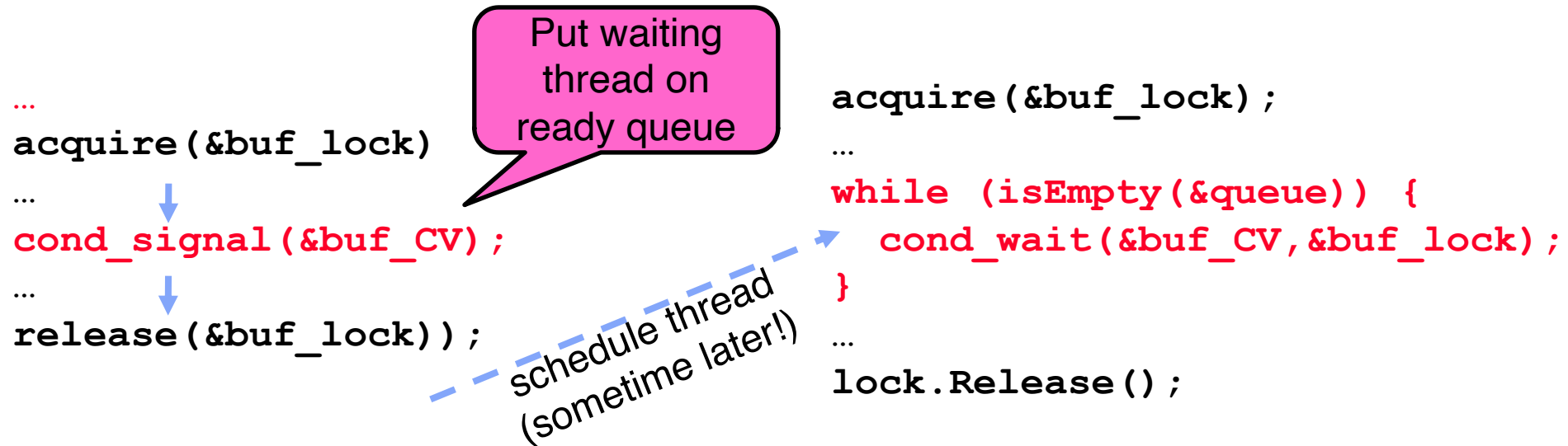
- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Then, Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again



- On first glance, this seems like good semantics
 - Waiter gets to run immediately, condition is still correct!
- Most textbooks talk about Hoare scheduling
 - However, hard to do, not really necessary!
 - Forces a lot of context switching (inefficient!)

Mesa monitors

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



- 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
- Most real operating systems do this!
 - More efficient, easier to implement
 - Signaler’s cache state, etc. still good

Circular Buffer – 3rd cut (Monitors, pthread-like)

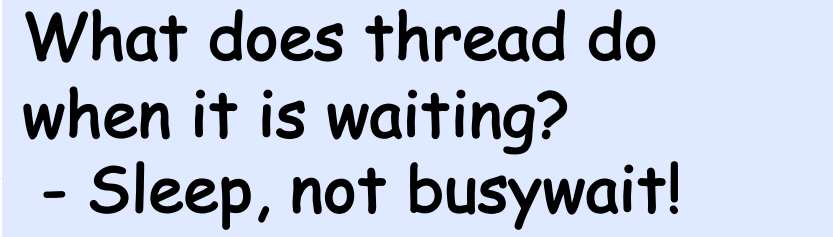
```
lock buf_lock = <initially unlocked>
```

```
condition producer_CV = <initially empty>
```

```
condition consumer_CV = <initially empty>
```

```
Producer(item) {  
    acquire(&buf_lock);  
    while (buffer full) { cond_wait(&producer_CV, &buf_lock); }  
    enqueue(item);  
    cond_signal(&consumer_CV);  
    release(&buf_lock);  
}
```

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

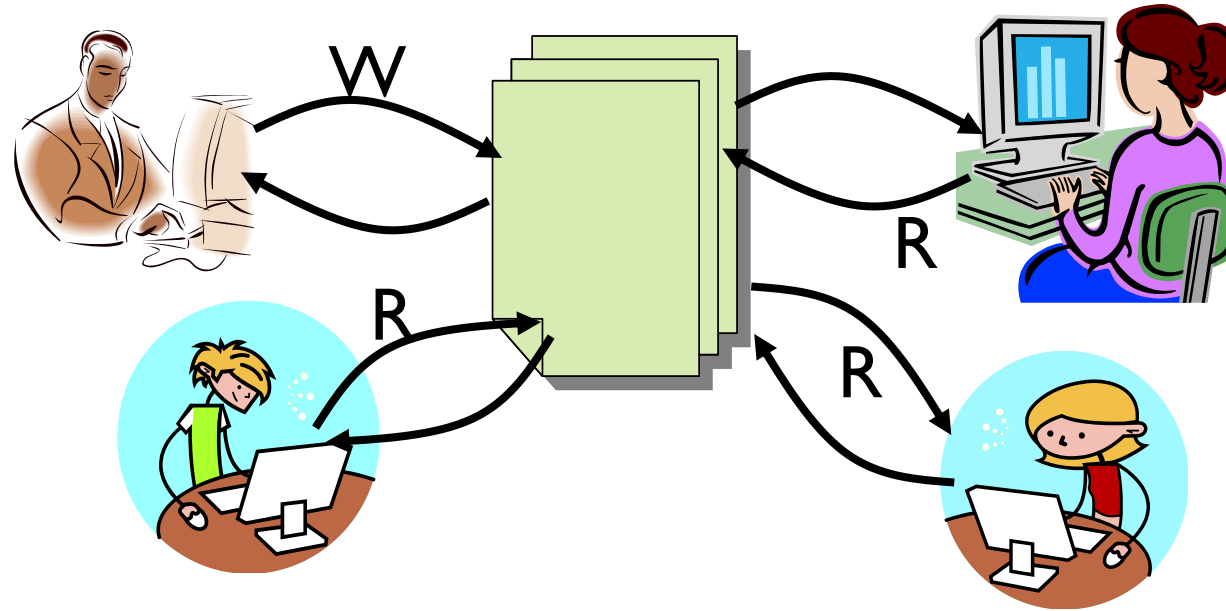


**What does thread do
when it is waiting?
- Sleep, not busywait!**

Group Discussion

- Topic: synchronization APIs
 - How to implement producer-consumer with a circular buffer with locks, semaphores and monitors?
 - What are the pros and cons of each solution?
- Discuss in groups of two to three students
 - Each group chooses a leader to summarize the discussion
 - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

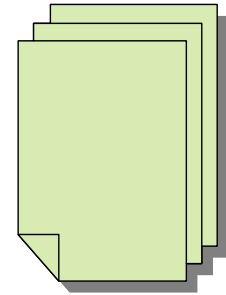
Readers/Writers Problem



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

Basic Readers/Writers Solution

- Correctness Constraints:
 - Readers can access database when no writers
 - Writers can access database when no readers or writers
 - Only one thread manipulates state variables at a time
- Basic structure of a solution:
 - **Reader()**
 - Wait until no writers
 - Access database
 - Check out - wake up a waiting writer
 - **Writer()**
 - Wait until no active readers or writers
 - Access database
 - Check out - wake up waiting readers or writer
 - State variables (Protected by a lock called “lock”):
 - » int AR: Number of active readers; initially = 0
 - » int WR: Number of waiting readers; initially = 0
 - » int AW: Number of active writers; initially = 0
 - » int WW: Number of waiting writers; initially = 0
 - » Condition okToRead = NIL
 - » Condition okToWrite = NIL



Code for a Reader

```
Reader() {
    // First check self into system
    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--;
if (AR == 0 && WW > 0)
```

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!
    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!
    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--;
    if (AR == 0 && WW > 0)
```

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 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

- R1 finishes (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

- 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) ;
}
```

Group Discussion

- 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
```

- 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()

Use of Single CV: okContinue

```
Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) {
        WR++;
        cond_wait(&okContinue, &lock);
        WR--;
    }
    AR++;
    release(&lock);

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okContinue);
    release(&lock);
}
```

```
Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) {
        WW++;
        cond_wait(&okContinue, &lock);
        WW--;
    }
    AW++;
    release(&lock);

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    acquire(&lock);
    AW--;
    if (WW > 0) {
        cond_signal(&okContinue);
    } else if (WR > 0) {
        cond_broadcast(&okContinue);
    }
    release(&lock);
}
```

**What if we turn okToWrite and okToRead into okContinue
(i.e. use only one condition variable instead of two)?**

Use of Single CV: okContinue

```
Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) {
        WR++;
        cond_wait(&okContinue,&lock);
        WR--;
    }
    AR++;
    release(&lock);

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okContinue);
    release(&lock);
}
```

```
Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) {
        WW++;
        cond_wait(&okContinue,&lock);
        WW--;
    }
    AW++;
    release(&lock);

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    acquire(&lock);
    AW--;
    if (WW > 0){
        cond_signal(&okContinue);
    } else if (WR > 0) {
        cond_broadcast(&okContinue);
    }
}
```

Consider this scenario:

- R1 arrives
- W1, R2 arrive while R1 still reading → W1 and R2 wait for R1 to finish
- Assume R1's signal is delivered to R2 (not W1)

Use of Single CV: okContinue

```
Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) {
        WR++;
        cond_wait(&okContinue,&lock);
        WR--;
    }
    AR++;
    release(&lock);

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_broadcast(&okContinue);
    release(&lock);
}
```

**Need to change to
broadcast()!**

```
Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) {
        WW++;
        cond_wait(&okContinue,&lock);
        WW--;
    }
    AW++;
    release(&lock);

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    acquire(&lock);
    AW--;
    if (WW > 0 || WR > 0){
        cond_broadcast(&okContinue);
    }
    release(&lock);
}
```

**Must broadcast()
to sort things out!**

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait(Semaphore *thesema) { semaP(thesema); }
Signal(Semaphore *thesema) { semaV(thesema); }
```
- Does this work better?

```
Wait(Lock *thelock, Semaphore *thesema) {
    release(thelock);
    semaP(thesema);
    acquire(thelock);
}
Signal(Semaphore *thesema) {
    semaV(thesema);
}
```

 - No: Condition vars have no history, semaphores have history:
 - » What if thread signals and no one is waiting?
 - » What if thread later waits?
 - » What if thread V's and no one is waiting?
 - » What if thread later does 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 *thelock, Semaphore *thesema) {
    release(thelock);
    semaP(thesema);
    acquire(thelock);
}
Signal(Semaphore *thesema) {
    if semaphore queue is not empty
        semaV(thesema);
}
```

 - 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
 - Can you come up with simpler Mesa-scheduled solution?

Mesa Monitor Conclusion

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

```
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
```



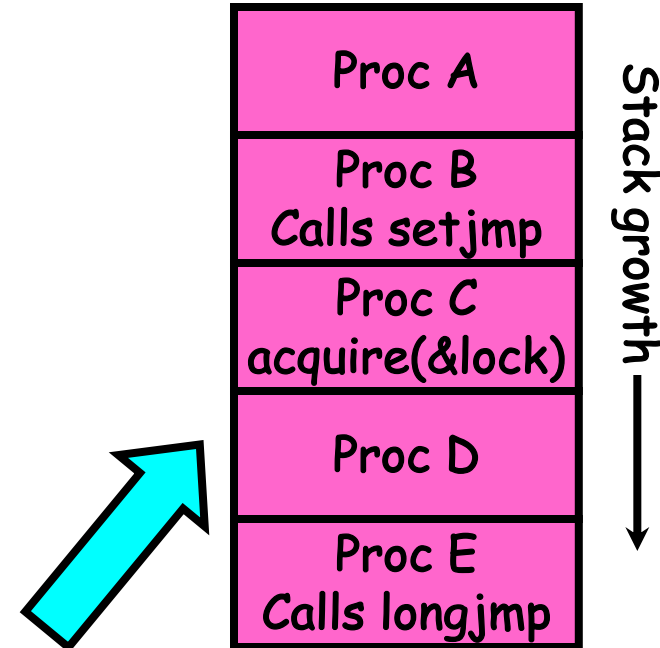
Check and/or update
state variables

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;  
}
```

- 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!



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();
}
```

- Is goto a solution???

```
void Rtn() {
    lock1.acquire();
    if (error) {
        goto release_lock1_and_return;
    }
    ...
    lock2.acquire();
    ...
    if (error) {
        goto release_both_and_return;
    }
    ...
release_both_and_return:
    lock2.release();
release_lock1_and_return:
    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)

– 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)

- 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();`

(OSDI 06) The Chubby lock service for loosely-coupled distributed systems

- Lock service
- Loosely-coupled distributed system
 - Coarse-grained synchronization
- UNIX-link file system interface
- Availability and reliability
- Open-source counterparts: Apache ZooKeeper, etcd

“Building Chubby was an engineering effort required to fill the needs mentioned above; it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it.”

Conclusion

- **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()**
- Monitors represent the logic of the program
 - Wait if necessary
 - Signal when change something so any waiting threads can proceed
 - Monitors supported natively in a number of languages
- Readers/Writers Monitor example
 - Shows how monitors allow sophisticated controlled entry to protected code