Synchronization 4: Semaphores (Con’t), Monitors and Readers/Writers

February 15th, 2022
Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Atomic Read-Modify-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;
}

- swap (address, register) { /* x86 */
  temp = M[address]; // swap register’s value to
  M[address] = register; // value at “address”
  register = temp;
}

- 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;
  }
}

- load-linked&store-conditional(address) { /* R4000, alpha */
  loop:
    ld r1, M[address];
    movi r2, 1; // Can do arbitrary computation
    sc r2, M[address];
    beqz r2, loop;
}
Recall: Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Mostly. Idea: only busy-wait to atomically check lock value

```c
int guard = 0; // Global Variable!
int mylock = FREE; // Interface: acquire(&mylock);
    // release(&mylock);

acquire(int *thelock) {
    // Short busy-wait time
    while (test&set(guard));
    if (*thelock == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
        // guard == 0 on wakup!
    } else {
        *thelock = BUSY;
        guard = 0;
    }
}

release(int *thelock) {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        *thelock = 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: Linux 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);
```

- **uaddr** points to a 32-bit value in user space
- **futex_op**
  - FUTEX_WAIT – if val == *uaddr sleep till FUTEX_WAIT
    » **Atomic** check that condition still holds after we disable interrupts (in kernel!)
  - FUTEX_WAKE – wake up at most val waiting threads
  - FUTEX_FD, FUTEX_WAKE_OP, FUTEX_CMP_REQUEUE: More interesting operations!
- **timeout**
  - ptr to a **timespec** structure that specifies a timeout for the op

- 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: Lock Using Atomic Instructions and Futex

- Three (3) states:
  - UNLOCKED: No one has lock
  - LOCKED: One thread has lock
  - CONTESTED: Possibly more than one (with someone sleeping)

- Clean interface!
- Lock grabbed cleanly by either
  - compare_and_swap()
  - First swap()

- No overhead if uncontested!
- Could build semaphores in a similar way!

```c
typedef enum { UNLOCKED, LOCKED, CONTESTED } Lock;
Lock mylock = UNLOCKED; // Interface: acquire(&mylock);
     //          release(&mylock);

acquire(Lock *thelock) {
    // If unlocked, grab lock!
    if (compare&swap(thelock, UNLOCKED, LOCKED))
        return;

    // Keep trying to grab lock, sleep in futex
    while (swap(mylock, CONTESTED) != UNLOCKED))
        // Sleep unless someone releases hear!
        futex(thelock, FUTEX_WAIT, CONTESTED);
}

release(Lock *thelock) {
    // If someone sleeping,
    if (swap(thelock, UNLOCKED) == CONTESTED)
        futex(thelock, FUTEX_WAKE, 1);
}
```
Recall: Producer-Consumer with a Bounded Buffer

- **Problem Definition**
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out
  - Need synchronization to coordinate producer/consumer

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

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

- **Example 2: Coke machine**
  - Producer can put limited number of Cokes in machine
  - Consumer can’t take Cokes out if machine is empty

- **Others: Web servers, Routers, …**
Recall: Circular Buffer Data Structure (sequential case)

```c
typedef struct buf {
    int write_index;
    int read_index;
    <type> *entries[BUFSIZE];
} buf_t;
```

- Insert: write & bump write ptr (enqueue)
- Remove: read & bump read ptr (dequeue)
- How to tell if Full (on insert) Empty (on remove)?
- And what do you do if it is?
- What needs to be atomic?
Recall: Circular Buffer – first cut

mutex buf_lock = <initially unlocked>

Producer(item) {
    acquire(&buf_lock);
    while (buffer full) {}; // Wait for a free slot
    enqueue(item);
    release(&buf_lock);
}

Consumer() {
    acquire(&buf_lock);
    while (buffer empty) {}; // Wait for arrival
    item = dequeue();
    release(&buf_lock);
    return item
}
mutex buf_lock = <initially unlocked>

Producer(item) {
    acquire(&buf_lock);
    while (buffer full) {release(&buf_lock); acquire(&buf_lock);}
    enqueue(item);
    release(&buf_lock);
}

Consumer() {
    acquire(&buf_lock);
    while (buffer empty) {release(&buf_lock); acquire(&buf_lock);}
    item = dequeue();
    release(&buf_lock);
    return item
}
Higher-level Primitives than Locks

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

• Good primitives and practices important!
  – Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  – UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs

• Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  – This lecture presents some ways to structuring sharing
Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: 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
- Technically examining value after initialization is not allowed.
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
- POSIX adds ability to read value, but technically not part of proper interface!
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:
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);
  ```

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);
  }
  ```
Revisit 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)

- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine

- General rule of thumb: Use a separate semaphore for each constraint
  - Semaphore fullBuffers; // consumer’s constraint
  - Semaphore emptyBuffers;// producer’s constraint
  - Semaphore mutex; // mutual exclusion
Full Solution to Bounded Buffer (coke machine)

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 is more coke
}

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

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)`

• Is order of P’s important?
  Yes! Can cause deadlock

• Is order of V’s important?
  No, except that it might affect scheduling efficiency

• What if we have 2 producers or 2 consumers?
  Decrease # of empty slots
  Increase # of occupied slots
  Decrease # of occupied slots
  Increase # of empty slots

```c
Producer(item) {
  semaP(&mutex);
  semaP(&emptySlots);
  Enqueue(item);
  semaV(&mutex);
  semaV(&fullSlots);
}

Consumer() {
  semaP(&fullSlots);
  semaP(&mutex);
  item = Dequeue();
  semaV(&mutex);
  semaV(&emptySlots);
  return item;
}
```
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
Administtrivia

- Midterm Thursday (February 17)!
  - No class on day of midterm
  - 7-9PM
  - All materials up to today’s lecture!
- Head TA will be posting where you are supposed to go
  - We have 3 primary rooms, and others
- If you are sick, let us know.
  - Do not come to the midterm!
- No class on Thursday
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!
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
Synchronized Buffer (with condition variable)

- Here is an (infinite) synchronized queue:

  ```
  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. Consider a piece of our dequeue code:

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

- Why didn’t we do this?

```c
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

```c
... acquire(&buf_lock);
... cond_signal(&buf_CV);
... release(&buf_lock));

acquire(&buf_lock);
... while (isEmpty(&queue)) {
    cond_wait(&buf_CV,&buf_lock);
}...
lock.Release();
```
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
}
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?
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 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
- Basic structure of mesa monitor-based program:

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

  do something so no need to wait

  lock

  condvar.signal();
  unlock
  ```
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);
}

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--; // No longer waiting
    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

```c
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--; // Now we are active!
    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

```c
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

```c
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);
}
```

```c
acquire(&lock);
AR--; // Now we are active!
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--;            // Now we are active!
    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--; // No. Readers exist
  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--; // No. Readers exist
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

```c
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);
}
```

```c
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

```c
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--; // Now we are active!
    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

```c
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--; // Access completed
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

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

```c
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        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

```c
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--; // No active writers
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

```c
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--; // Now we are active!
    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

```c
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

```c
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--; // Now we are active!
    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

```c
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--; // Now we are active!
    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--; // No longer waiting
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}

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); // Acquire lock
AR--; // Decrease active readers
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--; // Now we are active!
    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--; // No. Readers exist
    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
    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

```c
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);
}
```

```c
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

```c
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--;            // No longer waiting
    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

```c
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);
}
```

```c
acquire(&lock);
AW--; // No. Active users exist
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

```c
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--; // No longer waiting
  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

```c
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++; // No longer waiting
    release(&lock);
}

AccessDBase(ReadWrite);

acquire(&lock);
AW--; // No. Active users exist
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

```c
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

```c
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);
    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--; // Any remaining writers must give up
    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

```c
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--; // AR = 0
    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

```cpp
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--; // No. Readers exist
    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

```c
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--; // No. Readers exist
    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

```c
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

```c
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);
}
```

```c
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

```c
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--; // Now we are active!
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

```c
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:

```c
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?

```c
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()

```c
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: \texttt{okContinue}

\begin{verbatim}
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);
}
\end{verbatim}

\textbf{What if we turn \texttt{okToWrite} and \texttt{okToRead} into \texttt{okContinue} (i.e. use only one condition variable instead of two)?}
Use of Single CV: \texttt{okContinue}

\textbf{Reader()}
\begin{verbatim}
  // 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);
\end{verbatim}

\textbf{Writer()}
\begin{verbatim}
  // 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);
\end{verbatim}

\textbf{Consider this scenario:}
- R1 arrives
- W1, R2 arrive while R1 still reading $\rightarrow$ W1 and R2 wait for R1 to finish
- Assume R1’s signal is delivered to R2 (not W1)
Use of Single CV: \texttt{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);
}

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);
}

\textcolor{red}{Need to change to broadcast()!}

\textcolor{red}{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);
  }
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:

```java
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();

Check and/or update state variables
unlock
```
C-Language Support for Synchronization

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

![Diagram showing stack growth and procedure calls](image-url)
Concurrency and Synchronization in C

• Harder with more locks

```c
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???

```c
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:

```c++
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:
    ```cpp
    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)

```java
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();
Recall: User/Kernel Threading Models

Almost all current implementations

Simple One-to-One Threading Model

Many-to-One

Many-to-Many
Recall: Thread State in the Kernel

• For every thread in a process, the kernel maintains:
  – The thread’s TCB
  – A kernel stack used for syscalls/interrupts/traps
    » This kernel-state is sometimes called the “kernel thread”
    » The “kernel thread” is suspended (but ready to go) when thread is running in user-space

• Additionally, some threads just do work in the kernel
  – Still has TCB
  – Still has kernel stack
  – But not part of any process, and never executes in user mode
In Pintos, Processes are Single-Threaded

- Pintos processes have only one thread
- TCB: Single page (4 KiB)
  - Stack growing from the top (high addresses)
  - struct thread at the bottom (low addresses)
- **struct thread** defines the TCB structure *and* PCB structure in Pintos
(Aside): Linux “Task”

- Linux “Kernel Thread”: 2 pages (8 KiB)
  - Stack and thread information on opposite sides
  - Containing stack and thread information + process descriptor
- One task_struct per thread
Multithreaded Processes (not in Pintos)

• Traditional implementation strategy:
  – One PCB (process struct) per process
  – Each PCB contains (or stores pointers to) each thread’s TCB

• Linux’s strategy:
  – One task_struct per thread
  – Threads belonging to the same process happen to share some resources
    » Like address space, file descriptor table, etc.

• To what extent does this actually matter?
Aside: Polymorphic Linked Lists in C

- Many places in the kernel need to maintain a “list of X”
  - This is tricky in C, which has no polymorphism
  - Essentially adding an interface to a package
- In Linux and Pintos this is done by embedding a list_elem in the struct
  - Macros allow shift of view between object and list
  - You saw this in Homework 1
Kernel Structure So Far (1/3)
Kernel Structure So Far (2/3)
These two threads:
• Are used internally by the kernel
• Don’t correspond to any particular user thread or process
Recall: Scheduling

- Question: How is the OS to decide which of several tasks to take off a queue?
- Scheduling: deciding which threads are given access to resources from moment to moment
  - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access
- Next time: we dive into scheduling!
Recall: Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine.

![Diagram showing the relationship between the processor, virtual address, physical address, and memory.]

- Processor
  - Registers

- Virtual address

- Translator

- Physical address

- Memory
  - Page Table
    - <Frame Addr>
  - 0x000...
  - 0xFFF...
Understanding “Address Space”

• Page table is the primary mechanism
• Privilege Level determines which regions can be accessed
  – Which entries can be used
• System (PL=0) can access all, User (PL=3) only part
• Each process has its own address space
• The “System” part of all of them is the same

All system threads share the same system address space and same memory
Page Table Mapping (Rough Idea)

Translation Map 1
Translation Map 2
Physical Address Space

(user process view of memory)

1/15/2022
Joesph & Kubiatowicz CS162 © UCB Spring 2022
User Process View of Memory

Process Virtual Address Space

Kernel: 0xffffffff
Argv: 0x008048000
Stack: 0xc0000000
Heap: 0x08048000
User Data: 0x00000000
User Code: 0x00000000

Processor registers:
- sp
- ip

Physical Memory

Page Table

Page
Processor Mode (Privilege Level)

Process Virtual Address Space

Kernel

argv

Stack

Heap

User data

User code

Page Table

Physical Memory

CPL: 3 - user

0x00000000

0xc0000000 argv

0x08048000

0x08048000

0x00000000

Processor registers

sp

ip

CPL: 3 - user
Aside: x86 (32-bit) Page Table Entry

- Controls many aspects of access
- Later – discuss page table organization
  - For 32 (64?) bit VAS, how large? vs size of memory?
  - Used sparsely

Pintos: page_dir.c
User → Kernel

Process Virtual Address Space

Processor registers

sp

ip

CPL: 3 - user

kernel

argv

stack

heap

user data

user code

0xffffffff

0xc0000000

0x08048000

0x00000000

Page Table

u/s
User → Kernel

Process Virtual Address Space

Processor registers

CPL: 0 - sys

Page Table

Physical Memory

User code

User data

Heap

Kernel code

Kernel data

argv

Stack

Page
Page Table Resides in Memory*

* In the simplest case. Actually more complex. More later.
Kernel Portion of Address Space

- Kernel memory is mapped into address space of every process
- Contains the kernel code
  - Loaded when the machine booted
- Explicitly mapped to physical memory
  - OS creates the page table
- Used to contain all kernel data structures
  - Lists of processes/threads
  - Page tables
  - Open file descriptions, sockets, ttys, ...
- Kernel stack for each thread
1 Kernel Code, Many Kernel Stacks

Process Virtual Address Space

- Kernel code
- Kernel data
- Stack
- Heap
- User data
- User code

Physical Memory

Page Table

Processor registers
- sp
- ip

CPL: 0 - sys

PTBR:
Conclusion

- **Semaphores**: Like integers with restricted interface
  - Two operations:
    » `P()`: Wait if zero; decrement when becomes non-zero
    » `V()`: Increment and wake a sleeping task (if exists)
    » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint

- **Monitors**: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`

- 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