Monitors

Loïc Guégan

Adapted from J. Kubiatowicz @ 2010 UCB, O. J. Anshus and T. Larsen and P. Ha @ UiT,
A. S. Tanenbaum @ 2008, A. Silberschatz @ 2009)

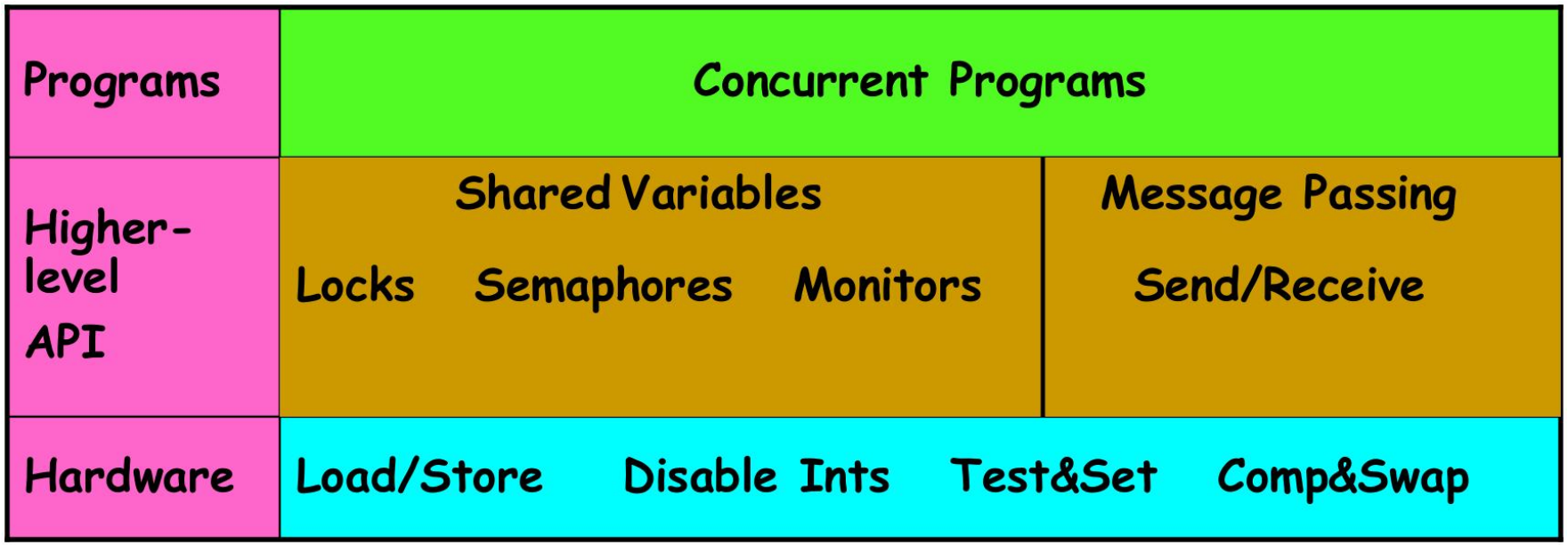
UiT The Arctic University of Norway

Spring - 2024

Outline

- **Motivation**
- Definition
- Monitor vs. semaphore
- Readers-Writers problem
  - A solution using monitors
- Language support for synchronization

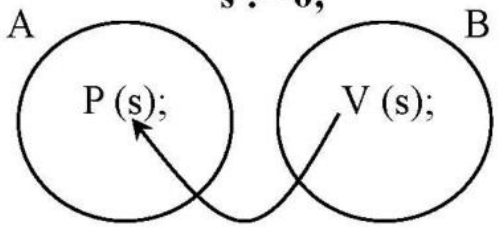
Overview of Synchronization Mechanisms



Review: semaphore

"The Signal"

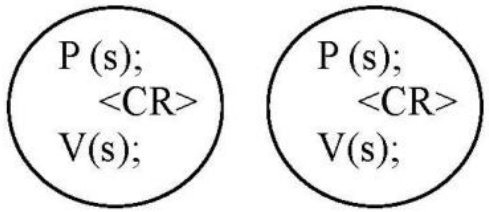
s := 0;



A blocks until B says V

"The Mutex"

s := 1;

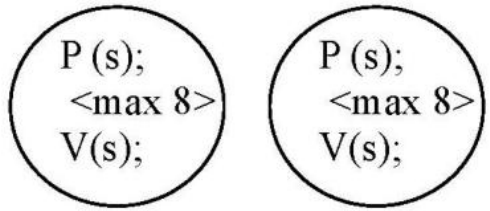


One thread gets in, next blocks until V is executed

NB: remember to set the initial semaphore value!

"The Team"

s := 8;



Up to 8 threads can pass P, the ninth will block until V is said by one of the eight already in there

Problems with Semaphores

Incorrect use of semaphore operations:

- V(mutex) ... P(mutex)
- P(mutex) ... V(mutex)
- Omitting of P(mutex) or V(mutex) (or both)

Recall: Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes 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: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty

Recall: Solution to Bounded Buffer

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

Producer(item) {
    emptyBuffers.P();         // Wait until space
    mutex.P();                // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();          // Tell consumers there is
                              // more item
}
Consumer() {
    fullBuffers.P();          // Check if there's an item
    mutex.P();                // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();         // tell producer need more
    return item;
}
```

Recall: Is the order of P's important?

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

Producer(item) {
    mutex.P();                 // Wait until buffer free
    emptyBuffers.P();          // Wait until space
    Enqueue(item);
    mutex.V();
    fullBuffers.V();           // Tell consumers there is
                               // more item
}
Consumer() {
    fullBuffers.P();           // Check if there's an item
    mutex.P();                 // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();          // tell producer need more
    return item;
}
```

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - Both mutual exclusion and scheduling constraints
    - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language

Outline

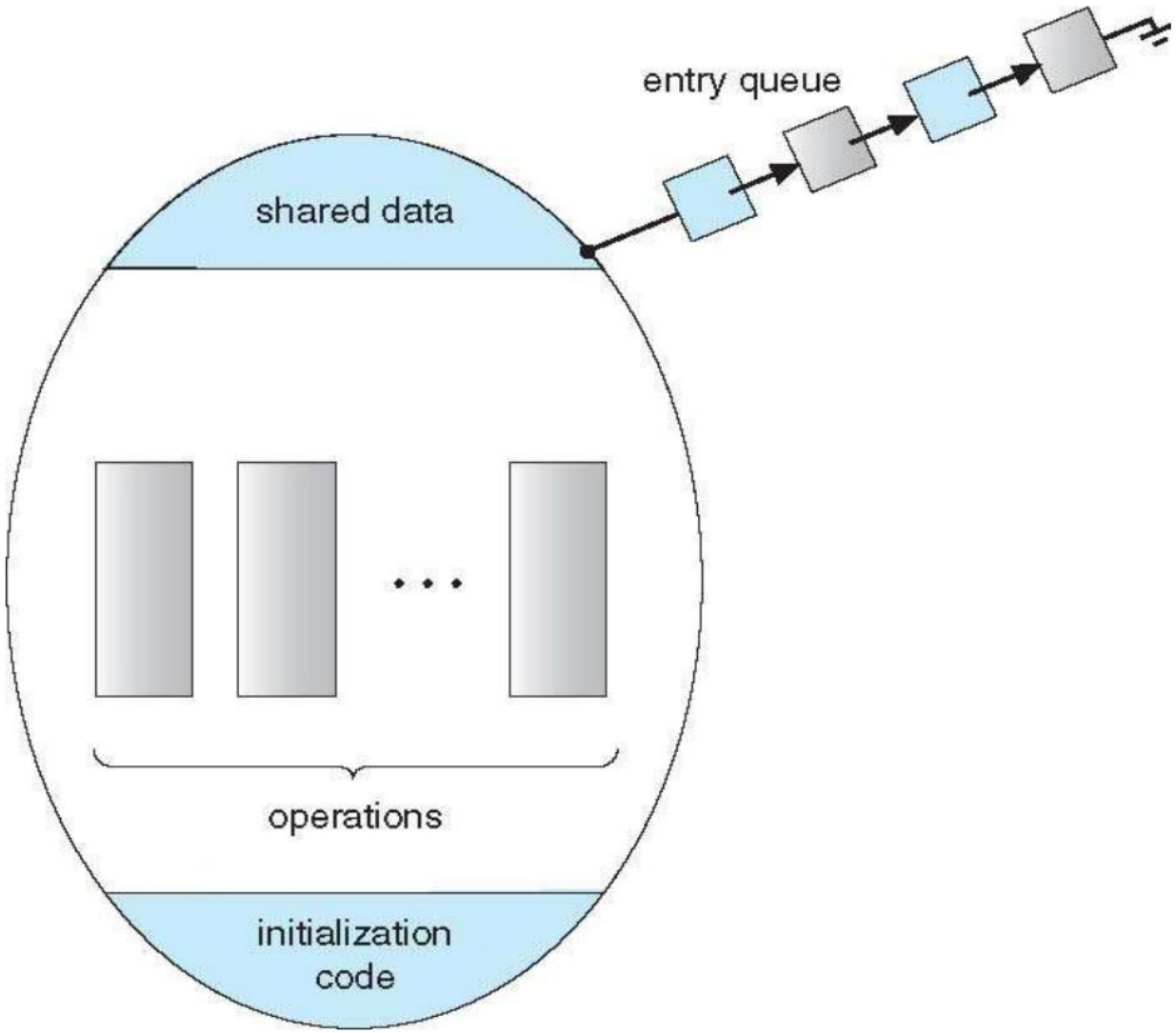
- Motivation
- **Definition**
- Monitor vs. semaphore
- Readers-Writers problem
  - A solution using monitors
- Language support for synchronization

Monitors

Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }
    ...
    procedure Pn (...) {......}
    Initialization code ( ....) { ... }

}
```

Schematic view of a Monitor



Simple Monitor Example (version 1)

- Producer-consumer with an infinite buffer

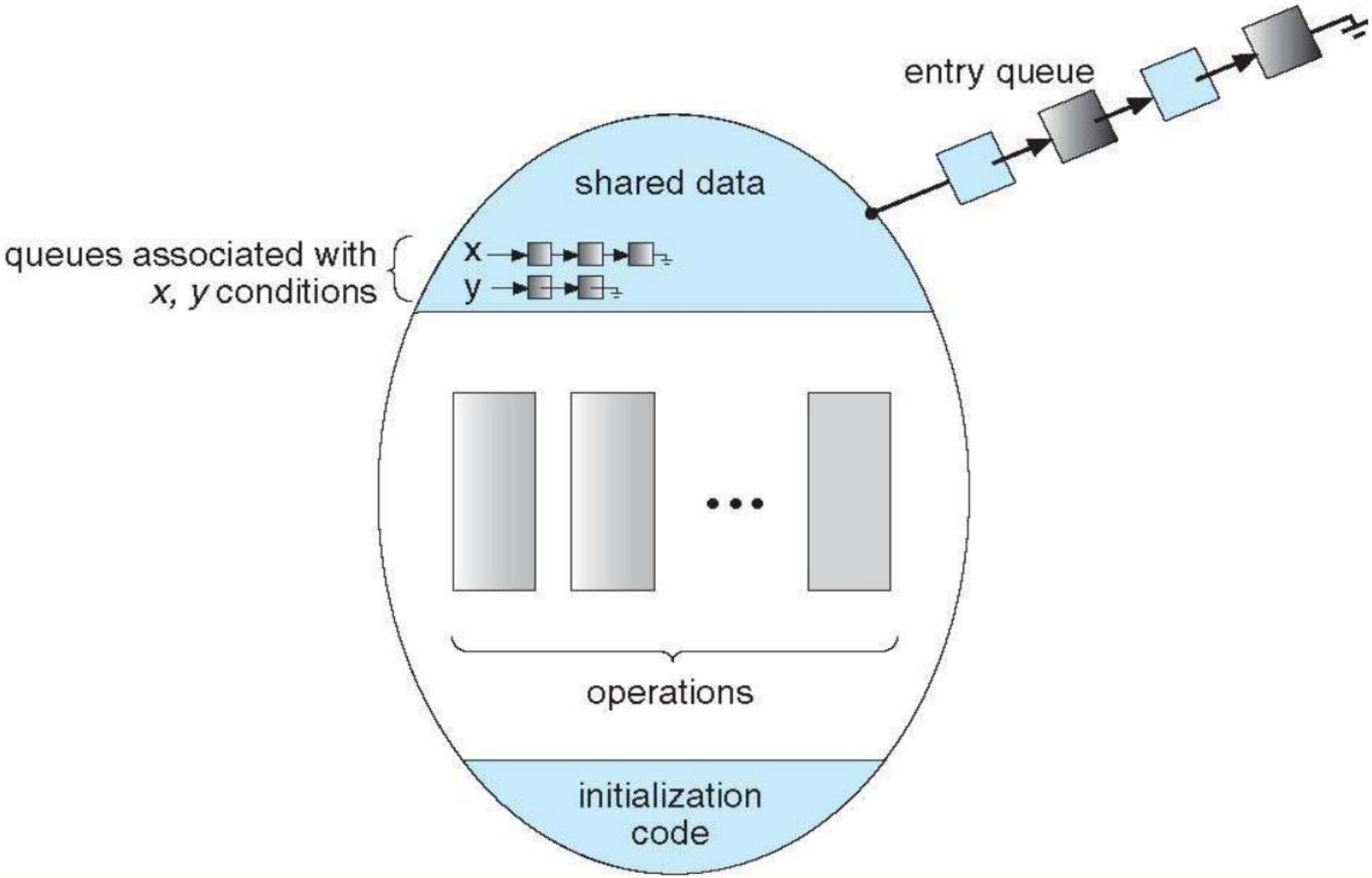
```
Lock lock;
Queue queue;

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

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

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

Condition Variables

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



Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

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

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

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

Mesa vs. Hoare monitors

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

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

  - Why didn't we do this?

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

- Answer: depends on the type of scheduling
  - Hoare-style (most textbooks):
    - Signaler gives lock, CPU to waiter; waiter runs immediately
    - Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  - Mesa-style (most real operating systems):
    - Signaler keeps lock and processor
    - Waiter placed on ready queue with no special priority
    - Practically, need to check condition again after wait

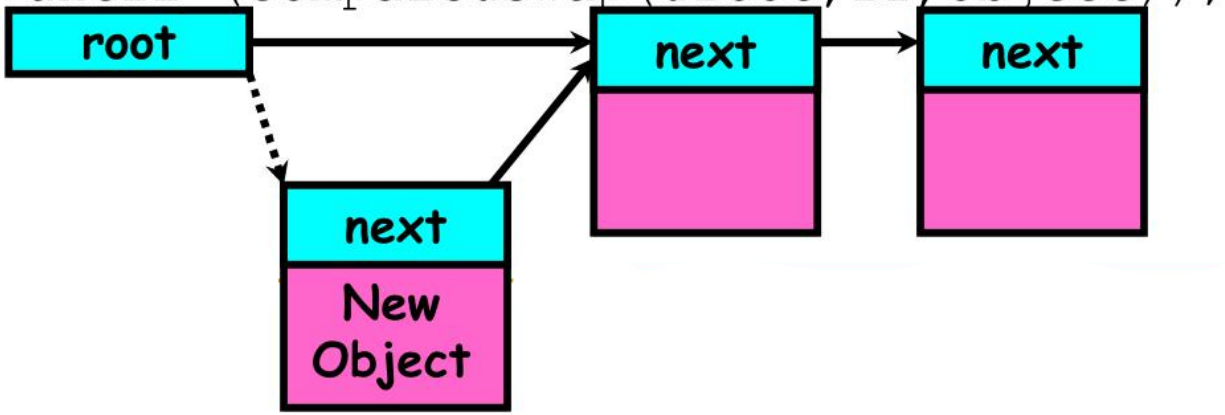
Using of Compare & Swap for queues

- compare&swap (&address, reg1, reg2)

```
atomic {
   if (reg1 == M[address]) {
      M[address] = reg2;
      return success;
   } else {
      return failure;
   }
}
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {
   do {                        // repeat until no conflict
      ld r1, M[root]           // Get ptr to current head
      st r1, M[object]         // Save link in new object
   } until (compare&swap(&root,r1,object));
}
```



Outline

- Motivation
- Definition
- **Monitor vs. semaphore**
- Readers-Writers problem
  - A solution using monitors
- Language support for synchronization

Can we construct Monitors from Semaphores?

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

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- Does this work better?

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

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

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

  - Not legal to look at contents of semaphore queue
  - 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 monitor in book Operating System Concepts by A. Silberschatz.

Semaphore vs. Monitor

Semaphore

- **P**(s) means WAIT if $s = 0$. Otherwise, $s = s - 1$
- **V**(s) means start a waiting thread and REMEMBER that a V call was made: $s = s + 1$
- Assume $s = 0$ when V(s) is called: If there is no thread to start this time, the next thread to call P(s) will get through P(s)

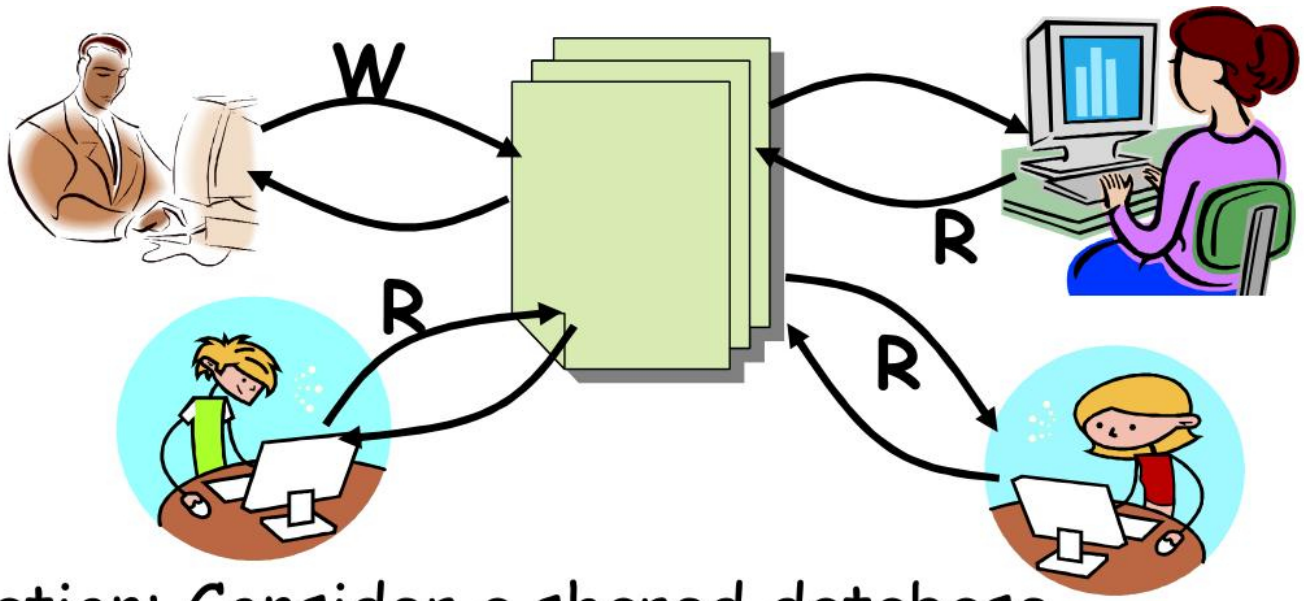
Monitor

- **Wait**(cond) means unconditional WAIT
- **Signal**(cond) means start a waiting thread. But no history!
- Assume that the condition queue is empty when Signal() is called. The next thread to call Wait(cond) will block.

Outline

- Motivation
- Definition
- Monitor vs. semaphore
- **Readers-Writers problem**
  - **A solution using monitors**
- Language support for synchronization

Readers/Writers Problem



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

Basic Readers/Writers Solution

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

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

  - `Writer()`

```
Wait until no active readers or writers
Access database
Check out - wake up waiting readers or writer
```

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

Code for a Reader

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

Code for a Writer

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

Simulation of Readers/Writers solution

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

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

- First, R1 comes along:
  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
  AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers may take a while to access database
  - Situation: Locks released

Simulation(2)

- Next, W1 comes along:

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

- Can't start because of readers, so go to sleep:
  AR = 2, WR = 0, AW = 0, WW = 1
- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:

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

Simulation(3)

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

```
if (WW > 0){              // Give priority to writers
  okToWrite.signal();     // Wake up one writer
} else if (WR > 0) {      // Otherwise, wake reader
  okToRead.broadcast(); // Wake all readers
}
```

  - Writer wakes up reader, so get:
    AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

Questions

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

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

- What if we erase the condition check in Reader exit?

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

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

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

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

Monitor Conclusion

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

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

do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables
Wait if necessary

Check and/or update state variables

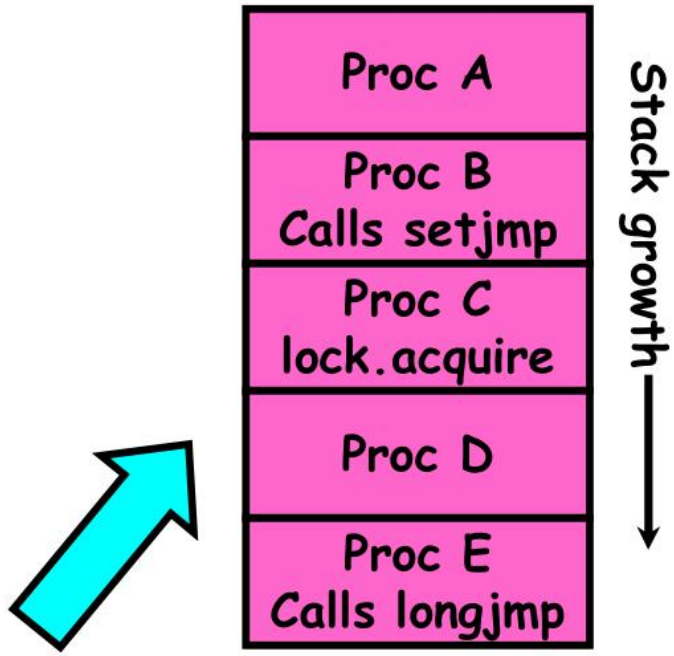
Outline

- Motivation
- Definition
- Monitor vs. semaphore
- Readers-Writers problem
  - A solution using monitors
- **Language support for synchronization**

C-Language Support for Synchronization

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

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



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

C++ 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++ 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;
   …
}
```

References

- A. S. Tanenbaum, Modern Operating Systems.
- A. Silberschatz et. al., Operating System Concepts.

Thanks for your attention!

Questions?