Goals for Today

- Continue with Synchronization Abstractions
  - Monitors and condition variables
- Readers-Writers problem and solution
- Language Support for Synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from lecture notes by Kubiatowicz.

Where are we going with synchronization?

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

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

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
Review: Semaphores

• Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  – P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    » Think of this as the wait() operation
  – V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    » This of this as the signal() operation
  – P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch

Correctness constraints for solution

• Correctness Constraints:
  – Consumer must wait for producer to fill slots, if empty (scheduling constraint)
  – Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)
  – Only one thread can manipulate buffer queue at a time (mutual exclusion)

• General rule of thumb:
  Use a separate semaphore for each constraint
  – Semaphore fullSlots; // consumer’s constraint
  – Semaphore emptySlots; // producer’s constraint
  – Semaphore mutex; // mutual exclusion

Full Solution to Bounded Buffer

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

Producer(item) {
    emptySlots.P(); // Wait until space
    mutex.P(); // Wait until slot free
    Enqueue(item);
    mutex.V();
    fullSlots.V(); // Tell consumers there is
                   // more coke
}

Consumer() {
    fullSlots.P(); // Check if there’s a coke
    item = Dequeue();
    mutex.V();
    emptySlots.V(); // tell producer need more
    return item;
}

Discussion about Solution

• Why asymmetry?
  – Producer does: emptySlots.P(), fullSlots.V()
  – Consumer does: fullSlots.P(), emptySlots.V()

• Is order of P’s important?
  Decrease # of empty slots  Increase # of occupied slots

• Is order of V’s important?
  Decrease # of occupied slots  Increase # of empty slots

• What if we have 2 producers or 2 consumers?
  – Do we need to change anything?
Solution for Bounded Buffer using Locks only

```c
int fullSlots = 0; // Initially, no coke
Lock lock = free; // No one using machine

Producer(item) {
    lock.Acquire();
    if (fullSlots == bufSize) {
        lock.Release(); // No room for coke
        return false;    // No room for coke
    }        // No room for coke
    Enqueue(item); // add new coke
    fullSlots++;   // add new coke
    lock.Release();
    return true;
}

Consumer() {
    lock.Acquire();
    if (fullSlots == 0) {
        lock.Release(); // no coke
        item = Dequeue(); // get coke
        fullSlots--;    // no coke
        lock.Release();  // no coke
        return item;
    }
    return null;   // no coke
}
```

Motivation for Monitors and Condition Variables

- Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints
- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

Monitor with Condition Variables

- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition Variable: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
Simple Monitor Example

- Here is an (infinite) synchronized queue
  
  ```
  Lock lock;
  Queue queue;
  
  AddToQueue(item) {
    lock.Acquire();       // Lock shared data
    queue.enqueue(item);  // Add item
    lock.Release();       // Release Lock
  }
  
  RemoveFromQueue() {
    lock.Acquire();       // Lock shared data
    item = queue.dequeue(); // Get next item or null
    lock.Release();       // Release Lock
  }
  ```

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

Condition Variables

- 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!

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
  }
  ```

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()) {
    item = queue.dequeue(); // Get next item
  }
  ```

- Why didn’t we do this?
  ```
  if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
  }
  ```

- Answer: depends on the type of scheduling
  - Hoare-style
  - Mesa-style
Hoare monitors

- Signaler gives up 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
- Most textbooks

```
Lock.Acquire()
...
dataready.signal();
...
lock.Release();
```

Mesa monitors

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

```
Lock.Acquire()
...
if (queue.isEmpty()) {
  dataready.wait(&lock);
}
...
lock.Release();
```

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
    lock.Acquire();
    while ({AW + WW} > 0) { // Is it safe to read?
        WR++;
        if (WR == 0) okToRead.wait(&lock); // Sleep on cond var
        WR--;
    }
    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++;
        if (WW == 0) okToWrite.wait(&lock); // Sleep on cond var
        WW--;
    }
    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();
}

Announcements

• Project 1 will be posted Thursday afternoon.

• Find a group or be dropped!

• Two new discussion section slots:
  – 5-6pm: 320 Soda Hall
  – 6-7pm: 320 Soda Hall

• We’ll announce by Friday which morning sections will move slots
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

```c
Reader() {
  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();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--; // No more readers exist
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}
```

Simulation of Readers/Writers Solution

• R1 comes along

  • AR = 1, WR = 0, AW = 0, WW = 0

```c
Reader() {
  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();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--; // No more readers exist
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}
```
Simulation of Readers/Writers Solution

- R1 comes along
  - AR = 1, WR = 0, AW = 0, WW = 0

Reader()
  lock.Acquire();
  while ((AW + WW) > 0) {
    // Is it safe to read?
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
  AR++; // Now we are active!
  lock.release();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();

Simulation of Readers/Writers Solution

- R2 comes along
  - AR = 1, WR = 0, AW = 0, WW = 0

Reader()
  lock.Acquire();
  while ((AW + WW) > 0) {
    // Is it safe to read?
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
  AR++; // Now we are active!
  lock.release();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();

Simulation of Readers/Writers Solution

- R2 comes along
- AR = 2, WR = 0, AW = 0, WW = 0

```c
Reader() {
    lock.acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++;
        okToRead.wait(&lock); // Sleep on cond var
        WR--;
    }
    AR++;
    lock.release();
    AccessDbase(ReadOnly);
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.signal();
    lock.release();
}
```

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() {
    lock.acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++;
        okToWrite.wait(&lock); // Sleep on cond var
        WW--;
    }
    AW++;
    lock.release();
    AccessDbase(ReadWrite);
    lock.acquire();
    AW--;
    if (WW > 0){
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
```
Simulation of Readers/Writers Solution

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

```java
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) {
        // Is it safe to write?
        if (AR > 0) {
            okToWrite.wait(&lock); // Sleep on cond var
            AR--;
        }
        AW++;
        lock.release();
    }
    AccessDbase(ReadWrite);
}
```

Simulation of Readers/Writers Solution

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

```java
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) {
        // Is it safe to write?
        if (AR > 0) {
            okToWrite.wait(&lock); // Sleep on cond var
            AR--;
        }
        AW++;
        lock.release();
    }
    AccessDbase(ReadWrite);
}
```

Simulation of Readers/Writers Solution

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

```java
W1 cannot start because of readers, so goes to sleep
```
Simulation of Readers/Writers Solution

R3 comes along (R1, R2 accessing dbase, W1 waiting)
AR = 2, WR = 0, AW = 0, WW = 1

Reader()
{ 
  lock.Acquire();
  while ((AW + WW) > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
  AR++;
  lock.release();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}

Simulation of Readers/Writers Solution

R3 comes along (R1, R2 accessing dbase, W1 waiting)
AR = 2, WR = 1, AW = 0, WW = 1

Reader()
{ 
  lock.Acquire();
  while ((AW + WW) > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
  AR++;
  lock.release();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}

Simulation of Readers/Writers Solution

R3 comes along (R1, R2 accessing dbase, W1 waiting)
AR = 2, WR = 0, AW = 0, WW = 1

Reader()
{ 
  lock.Acquire();
  while ((AW + WW) > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
  AR++;
  lock.release();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}

Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1, R3 waiting)
AR = 2, WR = 1, AW = 0, WW = 1

Reader()
{ 
  lock.Acquire();
  while ((AW + WW) > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
  AR++;
  lock.release();
  AccessDbase(ReadOnly);
  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
    okToWrite.signal();
  lock.Release();
}
Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1, R3 waiting)
- AR = 1, WR = 1, AW = 0, WW = 1

```c
Reader() {
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();
AccessDbase(ReadOnly);
lock.Acquire();
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();
}
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- AR = 1, WR = 1, AW = 0, WW = 1

```c
Reader() {
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();
AccessDbase(ReadOnly);
lock.Acquire();
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();
}
```
Simulation of Readers/Writers Solution

• R1 finishes (W1, R3 waiting)
• AR = 0, WR = 1, AW = 0, WW = 1

Reader()
lock.Acquire();
while ((AR + WR) > 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();
AccessDbase(ReadOnly);
lock.Acquire();
if (AR == 0 && WW > 0)
okToWrite.signal();
lock.Release();

W1 gets signal (R3 still waiting)

Writer()
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++; lock.release();
AccessDbase(ReadWrite);
lock.Acquire();
AW--; if (WW > 0) { okToWrite.signal();
else if (WR > 0) 
okToRead.broadcast(); }
lock.Release();

Got signal from R1

All reader finished, signal writer – note R3 still waiting

Page 13
Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

Writer()
lock.Acquire();
while ((AW + AR) > 0) { // Is it safe to write?
    okToWrite.wait(&lock); // No. Active users exist
    WW++; // No longer waiting
} AW++;
lock.release();
AccessDatabase(ReadWrite);

lock.Acquire();
if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
} lock.Release();

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

Writer()
lock.Acquire();
while ((AW + AR) > 0) { // Is it safe to write?
    okToWrite.wait(&lock); // No. Active users exist
    WW--; // No longer waiting
} AW++;
lock.release();
AccessDatabase(ReadWrite);

lock.Acquire();
if (WW > 0) {
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
} lock.Release();
Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```c
Writer() {
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        if (AW > 0) {
            okToWrite.wait(&lock); // No. Active users exist
            AW--;
        } else if (AW == 0) {
            lock.release();
        }
        AW++;
    }
    lock.release();
    AccessDbase(ReadWrite);
    lock.Acquire();
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

No waiting writer, signal reader R3

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- AR = 0, WR = 0, AW = 0, WW = 0

```c
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        if (AR > 0) {
            okToRead.signal();
            lock.release();
        } else if (AR == 0 && WW > 0) {
            okToRead.broadcast();
            lock.Release();
        }
        AR++;
    }
    lock.release();
    AccessDbase(ReadOnly);
    lock.Acquire();
    if (AR == 0 && WW > 0) {
        okToWrite.signal();
    }
    lock.Release();
}
```

Got signal from W1

DONE!
Read/Writer Questions

Reader()

// check into system
lock.Acquire();

while ((AW + WW) > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
}
AR++;
lock.release();

What if we remove this line?

// read-only access
AccessDbase(ReadOnly);

// check out of system
lock.Acquire();
AR--;
if (AR == 0 && WW > 0)
    okToWrite.signal();
lock.Release();

What if we turn signal to broadcast?

What if we turn okToWrite and okToRead into okContinue?

Writer()

// check into system
lock.Acquire();

while ((AW + AR) > 0) {
    WW++;
    okToWrite.wait(&lock);
    WW--;
}
AW++;
lock.release();

// read/write access
AccessDbase(ReadWrite);

// check out of system
lock.Acquire();
AW--;
if (WW > 0)
    okToWrite.signal();
else if (WR > 0)
    okToRead.broadcast();
lock.Release();

What if we turn okToWrite and okToRead into okContinue?

• R1 arrives
• W1, R2 arrive while R1 reads
• R1 signals R2
Read/Writer Questions

Reader() {
   // check into system
   lock.acquire();
   while ((AW + WR) > 0) {
      WR++;
      okContinue.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();
   // read-only access
   AccessDbase(ReadOnly);
   // check out of system
   lock.acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okContinue.broadcast();
   lock.release();
}

Writer() {
   // check into system
   lock.acquire();
   while ((AW + AR) > 0) {
      WW++;
      okContinue.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();
   // read/write access
   AccessDbase(ReadWrite);
   // check out of system
   lock.acquire();
   AW--;
   if (WW > 0)
      okContinue.signal();
   else if (WR > 0)
      okContinue.broadcast();
   lock.release();
}

Need to change to broadcast!
Why?

Summary

• 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()

• Readers/ Writers
  – Readers can access database when no writers
  – Writers can access database when no readers
  – Only one thread manipulates state variables at a time

• Language support for synchronization:
  – Java provides synchronized keyword and one condition-variable per object (with wait() and notify())