Module 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Java Synchronization
- Synchronization in Solaris 2
- Synchronization in Windows NT

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-butter problem (Chapter 4) has a race condition on the class data ***count***

Bounded Buffer

```
public class BoundedBuffer {
  public void enter(Object item) {
        // producer calls this method
  }


  public Object remove() {
        // consumer calls this method
  }
  // potential race condition on count
  private volatile int count;
}
```

enter() Method

```
// producer calls this method
public void enter(Object item) {
        while (count == BUFFER_SIZE)
                ; // do nothing
        // add an item to the buffer
        ++count;
        buffer[in] = item;
         in = (in + 1) % BUFFER_SIZE;
}
```

remove() Method

```
// consumer calls this method
   public Object remove() {
      Object item;
              while (count == 0)
                             ; // do nothing
               // remove an item from the buffer
               --count;
               item = buffer[out];
              out = (out + 1) % BUFFER_SIZE;
              return item;
   }
```

Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

Worker Thread

```
public class Worker extends Thread {
  public Worker(String n, int i, MutualExclusion s) {
    name = n;
    id = i;
    shared = s;
  }
  public void run() { /* see next slide */ }

  private String name;
  private int id;
  private MutualExclusion shared;
}
```

run() Method of Worker Thread

```
public void run() {
    while (true) {
        shared.enteringCriticalSection(id);
        // in critical section code
        shared.leavingCriticalSection(id);
        // out of critical section code
    }
}
```

MutualExclusion Abstract Class

```
public abstract class MutualExclusion {
   public static void criticalSection() {
            // simulate the critical section
   }

   public static void nonCriticalSection() {
            // simulate the non-critical section
   }
   public abstract void enteringCriticalSection(int t);
   public abstract void leavingCriticalSection(int t);
   public static final int TURN_0 = 0;
   public static final int TURN_1 = 1;
}
```

Testing Each Algorithm

```
public class TestAlgorithm
{
  public static void main(String args[]) {
        MutualExclusion alg = new Algorithm_1();

        Worker first = new Worker("Runner 0", 0, alg);
        Worker second = new Worker("Runner 1", 1, alg);

        first.start();
        second.start();
  }
}
```

Algorithm 1

```
public class Algorithm_1 extends MutualExclusion {
   public Algorithm_1() {
            turn = TURN_0;
   }
   public void enteringCriticalSection(int t) {
            while (turn != t)
                        Thread.yield();
   }
   public void leavingCriticalSection(int t) {
            turn = 1 - t;
   }
   private volatile int turn;
}
```

Algorithm 2

```
public class Algorithm_2 extends MutualExclusion  {
  public Algorithm_2() {
          flag[0] = false;
          flag[1] = false;
  }
  public void enteringCriticalSection(int t) {
          // see next slide
   }
   public void leavingCriticalSection(int t) {
           flag[t] = false;
  }

  private volatile boolean[] flag = new boolean[2];
}
```

Algorithm 2 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
   int other = 1 - t;
   flag[t] = true;
         while (flag[other] == true)
                 Thread.yield();
  }
```

Algorithm 3

```
public class Algorithm_3 extends MutualExclusion {
   public Algorithm_3() {
              flag[0] = false;
              flag[1] = false;
              turn = TURN_0;
   }
   public void enteringCriticalSection(int t) {/* see next slides */ }
   public void leavingCriticalSection(int t) {{/* see next slides */ }
   private volatile int turn;
   private volatile boolean[] flag = new boolean[2];
}
```

Algorithm 3 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
   int other = 1 - t;
   flag[t] = true;
   turn = other;

   while ( (flag[other] == true) && (turn == other) )
            Thread.yield();
  }
```

Algo. 3 – leavingingCriticalSection()

```
public void leavingCriticalSection(int t) {
        flag[t] = false;
}
```

Synchronization Hardware

```
public class HardwareData {
    public HardwareData(boolean v) {
            data = v;
    }
    public boolean get() {
            return data;
    }
    public void set(boolean v) {
            data = v;
    }
    private boolean data;
}
```

Test-and-Set Instruction (in Java)

```
public class HardwareSolution
{
   public static boolean testAndSet(HardwareData target) {
         HardwareData temp = new HardwareData(target.get());

         target.set(true);

         return temp.get();
   }
}
```

Thread using Test-and-Set

```
HardwareData lock = new HardwareData(false);

while (true) {
   while (HardwareSolution.testAndSet(lock))
         Thread.yield(); // do nothing
   // now in critical section code
   lock.set(false);
   // out of critical section
}
```

Swap instruction

```
public static void swap(HardwareData a, HardwareData b) {
   HardwareData temp = new HardwareData(a.get());
   a.set(b.get());
   b.set(temp.get());
}
```

Thread using Swap

```
HardwareData lock = new HardwareData(false);
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);
    do {
            HardwareSolution.swap(lock,key);
    } while (key.get() == true);
    // now in critical section code
    lock.set(false);
    // out of critical section
}
```


Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

P (*S*): **while** $S \leq 0$ **do** *no-op*;
S--;

V(*S*): *S*++;

Semaphore as General Synchronization Tool

Semaphore S; // initialized to 1

P(S);

CriticalSection()

V(S);

Semaphore Eliminating Busy-Waiting

```
P(S) {
   value--;
   if (value < 0) {
            add this process to list
            block
   }
}
V(S) {
   value++;
   if (value <= 0) {
            remove a process P from list
            wakeup(P);
   }
}
```

Synchronization Using Semaphores

```
public class FirstSemaphore {
  public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);
    Worker[] bees = new Worker[5];

    for (int i = 0; i < 5; i++)
      bees[i] = new Worker(sem, "Worker " + (new Integer(i)).toString() );

    for (int i = 0; i < 5; i++)
      bees[i].start();
  }
}
```

Worker Thread

```
public class Worker extends Thread {
   public Worker(Semaphore) { sem = s;}
   public void run() {
            while (true) {
                        sem.P();
                        // in critical section
                        sem.V();
                        // out of critical section
            }
   }
   private Semaphore sem;
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

P_0	P_1
$P(S)$;	$P(Q)$;
$P(Q)$;	$P(S)$;
$\vdots$	$\vdots$
$V(S)$;	$V(Q)$;
$V(Q)$	$V(S)$;

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore *S* as a binary semaphore.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

```
public class BoundedBuffer {
    public BoundedBuffer() { /* see next slides */ }
    public void enter() { /* see next slides */ }
    public Object remove() { /* see next slides */ }

    private static final int  BUFFER_SIZE = 2;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;
    private int in, out;
    private Object[] buffer;
}
```

Bounded Buffer Constructor

```
public BoundedBuffer() {
    // buffer is initially empty
    count = 0;
    in = 0;
    out = 0;
    buffer = new Object[BUFFER_SIZE];
    mutex = new Semaphore(1);
    empty = new Semaphore(BUFFER_SIZE);
    full = new Semaphore(0);
  }
```

enter() Method

```
public void enter(Object item) {
    empty.P();
    mutex.P();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    mutex.V();
    full.V();
}
```

remove() Method

```
public Object remove() {
    full.P();
    mutex.P();

    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.V();
    empty.V();

    return item;
  }
```

Readers-Writers Problem: Reader

```
public class Reader extends Thread {
   public Reader(Database db) {
            server = db;
   }
   public void run() {
    int c;
    while (true) {
            c = server.startRead();
            // now reading the database
             c = server.endRead();
    }
   }
   private Database  server;
}
```

Readers-Writers Problem: Writer

```
public class Writer extends Thread {
   public Writer(Database db) {
               server = db;
   }
   public void run() {
    while (true) {
               server.startWrite();
               // now writing the database
                server.endWrite();
    }
   }
   private Database  server;
}
```

Readers-Writers Problem (cont)

```
public class Database
{
   public Database() {
      readerCount = 0;
      mutex = new Semaphore(1);
      db = new Semaphore(1);
   }
    public int startRead() { /* see next slides */ }
    public int endRead()  { /* see next slides */ }
    public void startWrite() { /* see next slides */ }
    public void endWrite()  { /* see next slides */ }

    private int readerCount; // number of active readers
    Semaphore mutex;   // controls access to readerCount
    Semaphore db;         // controls access to the database
}
```

startRead() Method

```
public int startRead() {
    mutex.P();
    ++readerCount;

    // if I am the first reader tell all others
    // that the database is being read
    if (readerCount == 1)
        db.P();

    mutex.V();
    return readerCount;
}
```

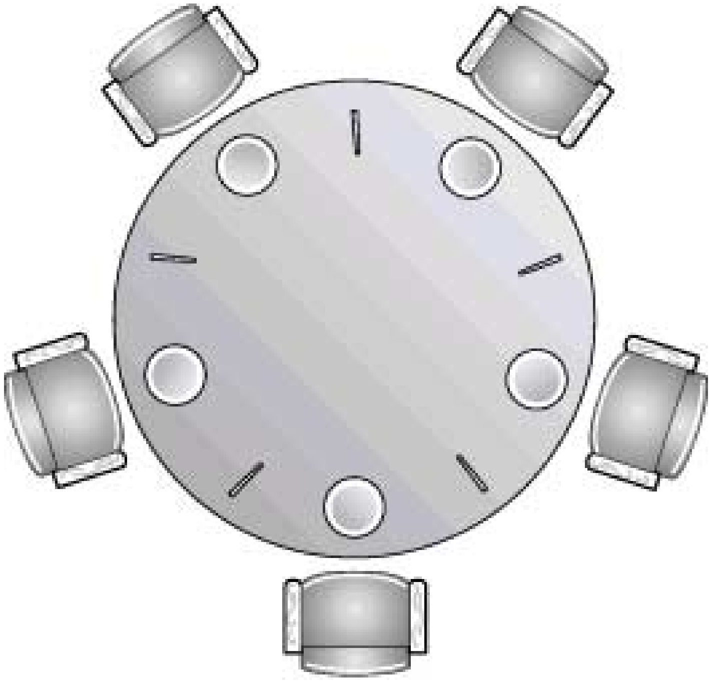
endRead() Method

```
public int endRead() {
    mutex.P();
    --readerCount;

    // if I am the last reader tell all others
    // that the database is no longer being read
    if (readerCount == 0)
      db.V();

    mutex.V();
    return readerCount;
  }
```

Writer Methods

```
public void startWrite() {
    db.P();
}

public void endWrite() {
    db.V();
}
```

Dining-Philosophers Problem



- Shared data

```
Semaphore chopStick[]  = new Semaphore[5];
```

Dining-Philosophers Problem (Cont.)

- Philosopher *i*:

```
while (true) {
    // get left chopstick
    chopStick[i].P();
    // get right chopstick
    chopStick[(i + 1) % 5].P();

    // eat for awhile

    //return left chopstick
     chopStick[i].V();
    // return right chopstick
     chopStick[(i + 1) % 5].V();

    // think for awhile
}
```

Monitors

- A monitor is a high-level abstraction that provides thread safety.
- Only one thread may be active within the monitor at a time.

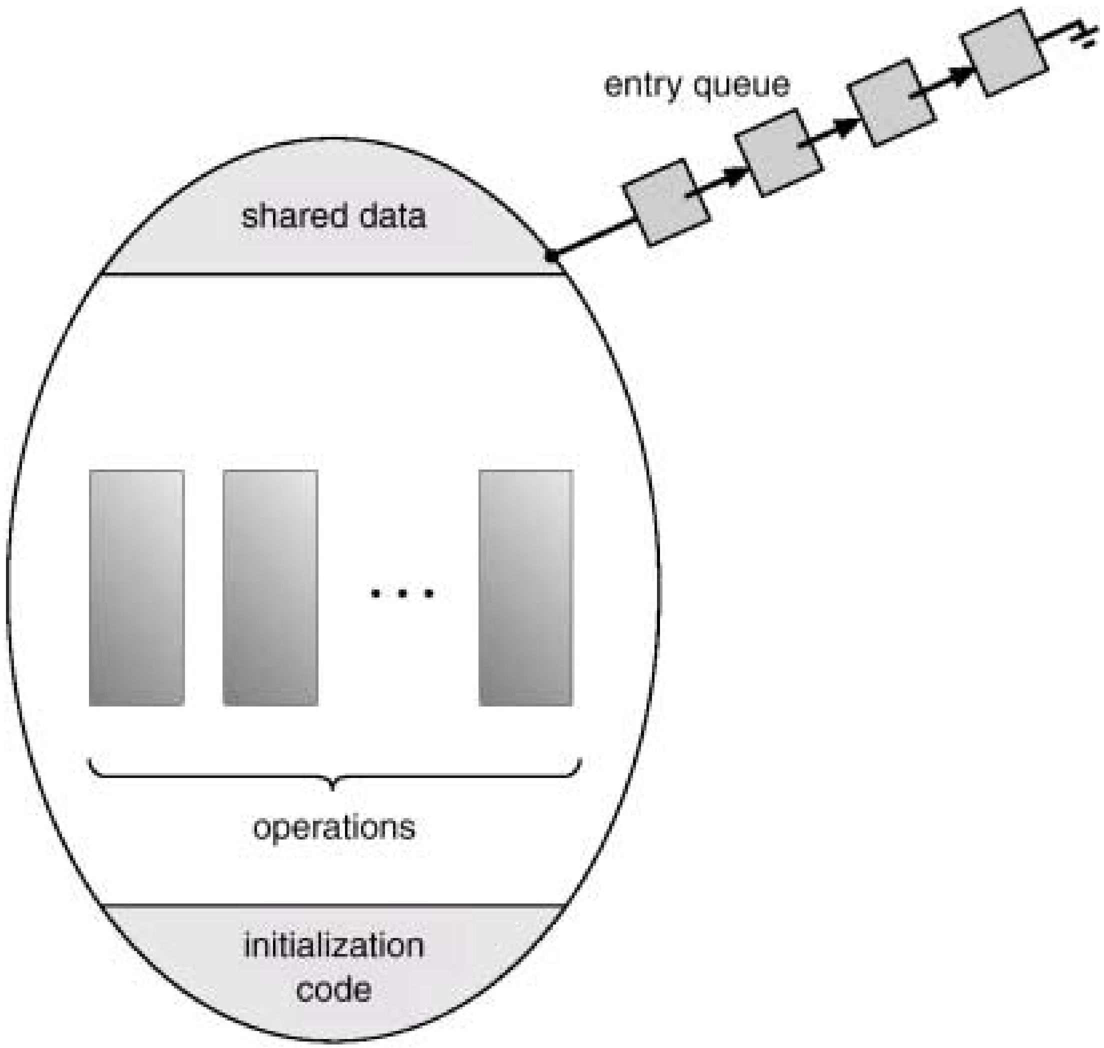
```
monitor monitor-name
{
   // variable declarations
   public entry p1(…) {
            …
   }
   public entry p2(…) {
            …
   }
}
```

Condition Variables

- condition x, y;

- A thread that invokes x.wait is suspended until another thread invokes x.signal

Monitor with condition variables



Solution to Dining Philosophers

```
monitor diningPhilosophers {
   int[] state = new int[5];
   static final int THINKING = 0;
   static final int HUNGRY = 1;
   static final int EATING = 2;
   condition[] self = new condition[5];
   public diningPhilosophers {
            for (int i = 0; i < 5; i++)
                        state[i] = THINKING;
   }
   public entry pickUp(int i) { /* see next slides */ }
   public entry putDown(int i) { /* see next slides */ }
   private test(int i) {/* see next slides */ }
}
```

pickUp() Procedure

```
public entry pickUp(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
                self[i].wait;
}
```

putDown() Procedure

```
public entry putDown(int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
}
```

test() Procedure

```
private test(int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING;
                self[i].signal;
        }
  }
```

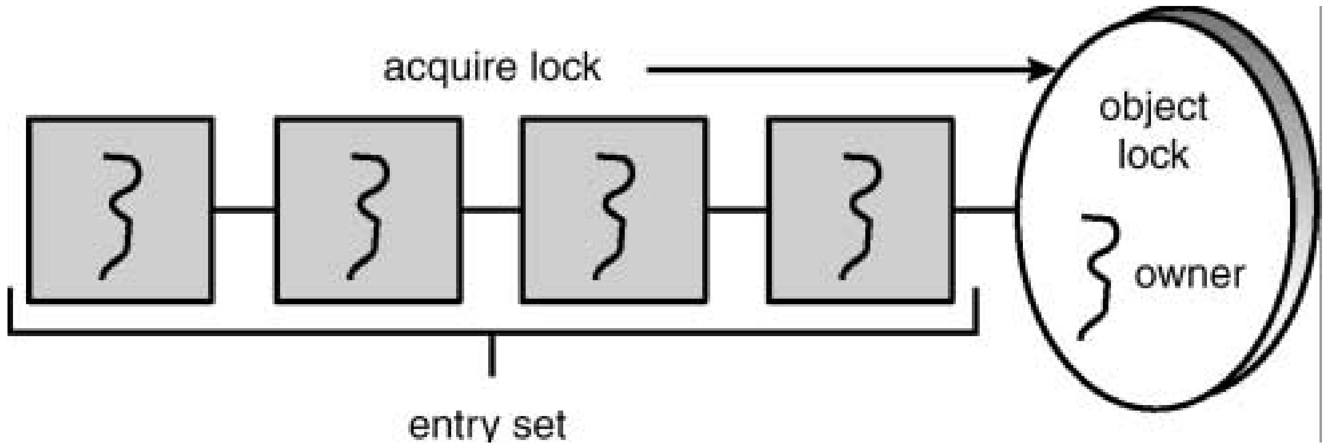
Java Synchronization

- Synchronized, wait(), notify() statements
- Multiple Notifications
- Block Synchronization
- Java Semaphores
- Java Monitors

synchronized Statement

- Every object has a lock associated with it.
- Calling a synchronized method requires “owning” the lock.
- If a calling thread does not own the lock (another thread already owns it), the calling thread is placed in the wait set for the object’s lock.
- The lock is released when a thread exits the synchronized method.

Entry Set



synchronized enter() Method

```
public synchronized void enter(Object item) {
   while (count == BUFFER_SIZE)
            Thread.yield();
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

synchronized remove() Method

```
public synchronized Object remove() {
   Object  item;
   while (count == 0)
            Thread.yield();
   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   return item;
}
```

The wait() Method

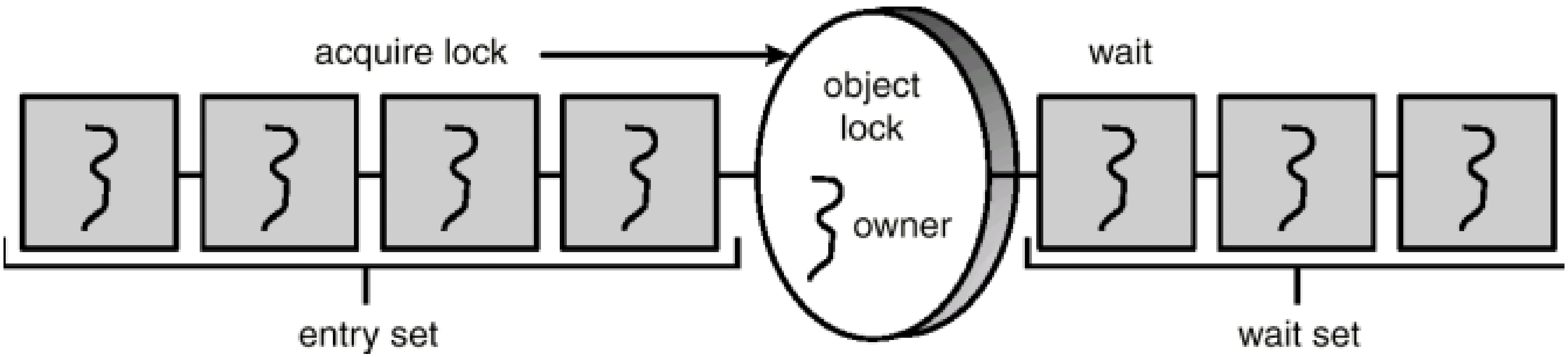
- When a thread calls wait(), the following occurs:

  - the thread releases the object lock.

  - thread state is set to blocked.

  - thread is placed in the wait set.

Entry and Wait Sets



The notify() Method

- When a thread calls notify(), the following occurs:

  - selects an arbitrary thread *T* from the wait set.

  - moves *T* to the entry set.

  - sets *T* to Runnable.

T can now compete for the object's lock again.

enter() with wait/notify Methods

```
public synchronized void enter(Object item) {
   while (count == BUFFER_SIZE)
            try {
                     wait();
            }
            catch (InterruptedException e) { }
   }
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   notify();
}
```

remove() with wait/notify Methods

```
public synchronized Object remove() {
    Object  item;
    while (count == 0)
            try {
                    wait();
            }
            catch (InterruptedException e) { }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();
    return item;
}
```

Multiple Notifications

- notify() selects an arbitrary thread from the wait set. *This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected.
- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set.

Reader Methods with Java Synchronization

```
public class Database {
   public Database() {
      readerCount = 0;
      dbReading = false;
      dbWriting = false;
   }
    public synchronized int startRead() { /* see next slides */ }
    public synchronized int endRead()  { /* see next slides */ }
    public synchronized void startWrite() { /* see next slides */ }
    public synchronized void endWrite()  { /* see next slides */ }

    private int readerCount;
    private boolean dbReading;
    private boolean dbWriting;
}
```

startRead() Method

```
public synchronized int startRead() {
   while (dbWriting == true) {
         try {
               wait();
         }
         catch (InterruptedException e) { }
         ++readerCount;
         if (readerCount == 1)
               dbReading = true;
         return readerCount;
}
```

endRead() Method

```
public synchronized int endRead() {
    --readerCount
    if (readerCount == 0)
      db.notifyAll();
    return readerCount;
  }
```

Writer Methods

```
public void startWrite() {
      while (dbReading == true || dbWriting == true)
            try {
                  wait();
            }
            catch (InterruptedException e) { }
            dbWriting = true;
}

public void endWrite() {
    dbWriting = false;
    notifyAll();

}
```

Block Synchronization

- Blocks of code – rather than entire methods – may be declared as synchronized.
- This yields a lock scope that is typically smaller than a synchronized method.

Block Synchronization (cont)

```
Object mutexLock = new Object();
. . .
public void someMethod() {
   // non-critical section
   synchronized(mutexLock) {
         // critical  section
   }
   // non-critical section
}
```

Java Semaphores

- Java does not provide a semaphore, but a basic semaphore can be constructed using Java synchronization mechanism.

Semaphore Class

```
public class Semaphore {
   public Semaphore() {
               value = 0;
    }
   public Semaphore(int v) {
               value = v;
    }
   public synchronized void P() { /* see next slide */ }
    public synchronized void V() { /* see next slide */ }
   private int value;
}
```

P() Operation

```
public synchronized void P() {
    while (value <= 0) {
      try {
        wait();
      }
      catch (InterruptedException e) { }
    }
    value --;
  }
```

V() Operation

```
public synchronized void V() {
    ++value;


    notify();
  }
```

Solaris 2 Synchronization

- Solaris 2 Provides:

  - adaptive mutex

  - condition variables

  - semaphores

  - reader-writer locks

Windows NT Synchronization

- Windows NT Provides:

  - mutex

  - critical sections

  - semaphores

  - event objects