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();
}
```


Silberschatz, Galvin, and Gagne ©1999

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;
  }
```

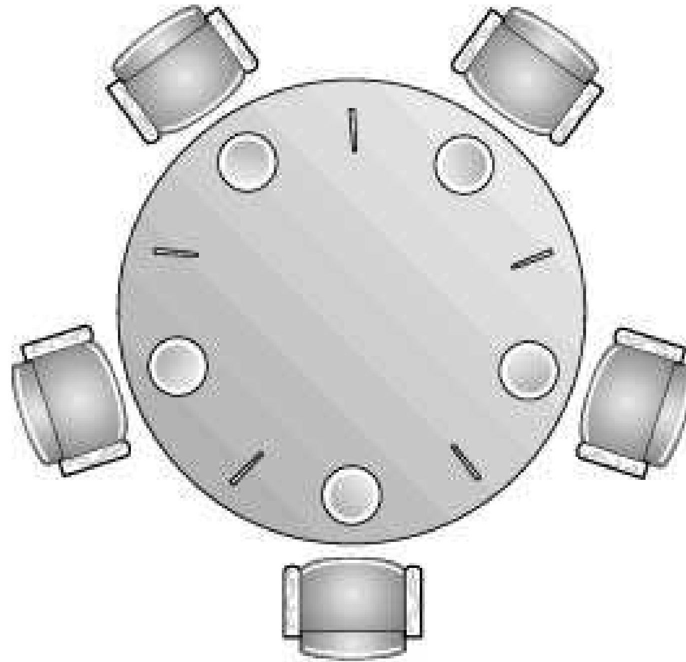
endRead() Method

```
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
}
```

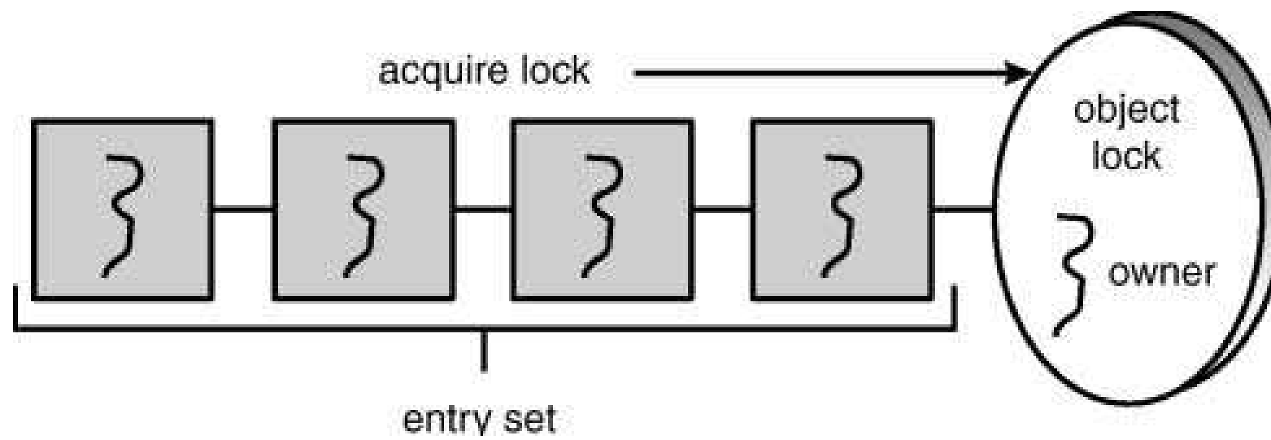
Java Synchronization

- 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

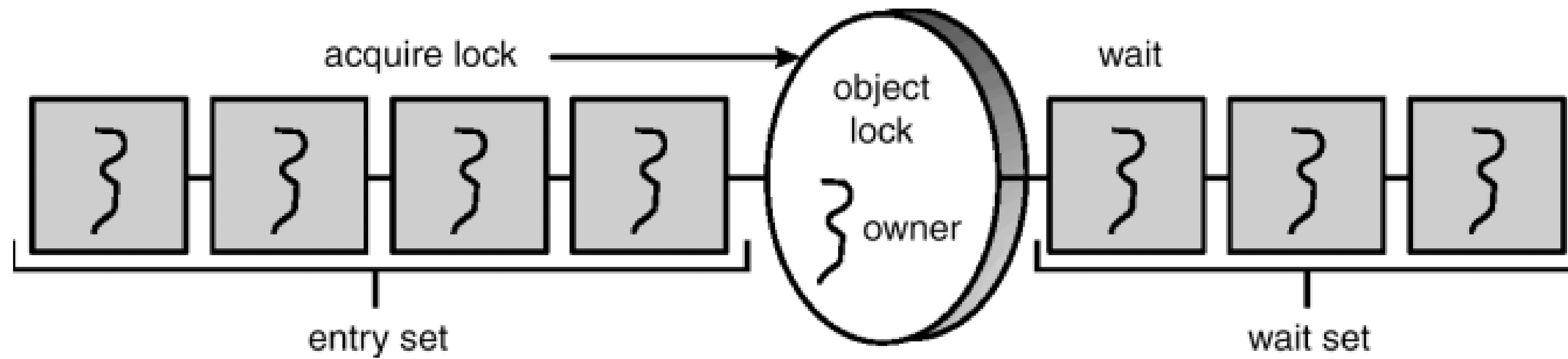
- When a thread calls wait(), the following occurs:

  - 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();
  }
```

Two brothers, Joe and John, share a common bank account, and can, independently, read the balance, make a deposit, and withdraw some money.

We can model the bank account as an object of the following class:

```
class Account {
    private double balance;

    public Account(double initialDeposit) {
        balance = initialDeposit;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        balance += amount;
   }

    public void withdraw(double amount) {
        if ( balance >= amount ) { balance -= amount; }
   }    // no negative balance allowed

}
```

We can model the account holders as two threads:

```
class AccountHolder extends Thread {
    private Account acc;

    public AccountHolder(Account a) {
        acc = a;
    }

   public void run() {
       ...
       acc.withdraw(100);
       ...
   }
}
```

Creation of the account and of the two account holders:

```
...
Account accnt = new Account(150);
AccountHolder John = new AccountHolder(accnt);
AccountHolder Joe  = new AccountHolder(accnt);
John.start();  Joe.start()
...
```

Note: John and Joe will share the same account which `accnt` refers to.

Example of interference

```
At the beginning: balance is 150

   John: accnt.withdraw(100)     Joe: accnt.withdraw(100)

   balance >= 100 ?
            |
            | yes                 balance >= 100 ?
            |                                |
            V                                | yes
   balance = balance - 100                   |
                                             V
                                  balance = balance - 100

At the end: balance is -50.
```

There is a problem also with the deposit method.

```
public void deposit(double amount) {
    balance += amount;
}
```

```
At the beginning: balance is 150

   John: accnt.deposit(100)     Joe: accnt.deposit(100)

   b = balance  (b = 150)
     |
     |                            b = balance  (b = 150)
     V                              |
   b += 100     (b = 250)           |
     |                              V
     |                            b += 100     (b = 250)
     V                              |
   balance = b                      |
                                    V
                                  balance = b

At the end: balance is 250 (instead of 350).
```

Note that there is a problem even if one of the methods updates the field and the other only reads the field. In general reading a field that is being updated by another thread might give unpredictable results.

There is no problem, on the contrary, with methods accessing common fields only in reading mode. Such read-only methods can (and should) run concurrently, thus optimizing execution time.

We have to incapsulate the critical sections so that they will be executed "atomically", i.e. without interleaving.

```
John: accnt.deposit(100)         Joe: accnt.deposit(100)
+---------------------------+
|                           |     delay until the critical section
| b = balance    (b = 150)  |     executed by John is completed
|   |                       |        ...
|   V                       |
| b += 100       (b = 250)  |
|   |                       |
|   V                       |        ...
| balance = b               |
+---------------------------+     John has terminated, Joe can start
                                 +---------------------------+
                                 |                           |
                                 | b = balance    (b = 250)  |
                                 |   |                       |
                                 |   V                       |
                                 | b += 100       (b = 350)  |
                                 |   |                       |
                                 |   V                       |
                                 | balance = b               |
                                 +---------------------------+
```

In Java, we can specify that we want a method to be executed atomically by declaring it synchronized.

Synchronized methods are mutually exclusive, i.e. the actions of two threads executing synchonized methods on the same objects cannot be interleaved

Example: the correct definition of Account

```
class Account {
    private double balance;

    public Account(double initialDeposit) {
        balance = initialDeposit;
    }

    public synchronized double getBalance() {
            ------------
        return balance;
    }

    public synchronized void deposit(double amount) {
            ------------
        balance += amount;
   }

    public synchronized void withdraw(double amount) {
            ------------
        if ( balance >= amount ) { balance -= amount; }
   }    // no negative balance allowed
}
```

```
 John: accnt.deposit(100)
                                Joe: accnt.deposit(100)
 acquire the lock on accnt
+---------------------------+     try to acquire the lock on accnt
|                           |     since the lock is not available,
| b = balance    (b = 150)  |     go in the "waiting list" of accnt
|   |                       |
|   |                       |        ... wait ...
|   V                       |
| b += 100       (b = 250)  |
|   |                       |
|   |                       |
|   V                       |
| balance = b               |        ... wait ...
|                           |
|                           |
```

```
|                         |
|                         |      ... wait ...
|                         |
|                         |
+-------------------------+
 release the lock on accnt

                              acquire the lock on accnt
                             +-------------------------+
                             |                         |
                             | b = balance    (b = 250) |
                             |   |                     |
                             |   |                     |
                             |   V                     |
                             | b += 100       (b = 350) |
                             |   |                     |
                             |   |                     |
                             |   V                     |
                             | balance = b             |
                             |                         |
                             +-------------------------+
                              release the lock on accnt
```

A few remarks about the locks

- Non synchronized methods do not require the lock, hence they can always be interleaved with each other and also with synchronized methods.
- A lock is "per thread". This means that nested invokations of synchronized methods on the same object will proceed without blocking.
- More precisely, there is a counter associated to a lock, which is incremented each time the thread enters a synchronized method on the object, and decremented when it completes a synchronized method. (It is the "level of nesting")

```
class C {

    ...

    public synchronized m1() {
           ------------
        ... m2(); ...;
    }

    public synchronized m2(){
           ------------
        ...
   }

}
```

obj.m1()

if obj.lock == 0
  then obj.lock = 1 (acquire lock)
  else go to wait list of obj