1

CSMC 412

Operating Systems
Prof. Ashok K Agrawala

Set 8

2

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();
  }
}
```


3

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;
}
```


4

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*);
⋮	⋮
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.

5

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.

6

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;
  }
```

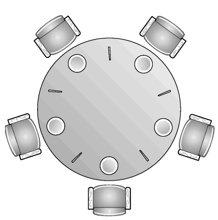

endRead() Method

```
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

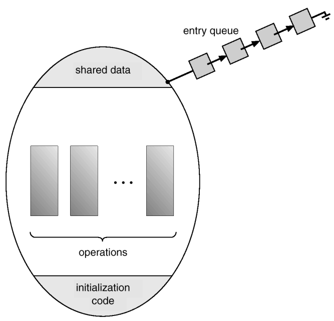
- A monitor is a high-level abstraction that provides thread safety.
- Only one thread may be active within the monitor at a time.

```
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;
        }
  }
```


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;
        }
  }
```

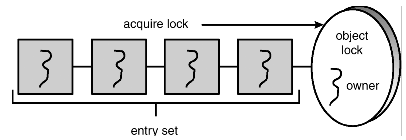

Java Synchronization

- 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;
}
```

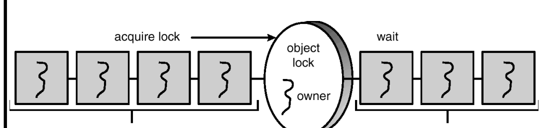

synchronized remove() Method

```
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

43

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

Spring 2001

Writer Methods

44

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

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

}
```

Spring 2001

Block Synchronization

45

- 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.

Spring 2001

Block Synchronization (cont)

46

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

Spring 2001

Java Semaphores

47

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

Spring 2001

Semaphore Class

48

```
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;
}
```

Spring 2001

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